

BOOK 7: SECURITY AND AUDITING

Chapter 1: Understanding Smart Contract Vulnerabilities

Smart contracts hold billions of dollars. They execute automatically. They cannot be stopped once deployed. One bug can drain everything. This chapter teaches you every vulnerability that hackers exploit so you can write code that survives attack.

SECTION 1: THE SECURITY MINDSET

Why Smart Contract Security Is Different

Traditional software has patches. You find a bug, you fix it, you deploy an update. Users upgrade. Problem solved. Smart contracts are different. Once deployed, the code is permanent. Immutable. If your contract has a vulnerability, attackers will find it. They will exploit it. You cannot patch it. You can only watch your funds drain.

This permanence changes everything about how you write code:

Traditional Development:

Move fast. Ship quickly. Fix bugs later. Users tolerate crashes. Data can be restored from backups. Worst case you lose some user trust.

Smart Contract Development:

Every line of code must be perfect before deployment. There are no backups. There is no restore. One mistake means permanent loss. Attackers have financial incentive to find your bugs. They will spend weeks analyzing your code. They have automated tools scanning every new contract.

The numbers tell the story. In 2023 alone, over 2 billion dollars was stolen from smart contracts. Not from phishing. Not from social engineering. From exploiting bugs in code that developers thought was secure.

The Attacker Mindset

To write secure code, you must think like an attacker. Ask yourself:

What assumptions does my code make?

What happens if those assumptions are wrong?

Can external contracts behave unexpectedly?

Can users provide malicious input?

What if functions are called in unexpected order?

What if transactions are reordered by miners?

What if multiple transactions interact?

Attackers do not use your contract the way you intended. They call functions with extreme values. They call them in wrong order. They call them from malicious contracts. They sandwich your transactions. They flash loan billions of dollars to manipulate prices. They do everything you did not anticipate.

SECTION 2: REENTRANCY ATTACKS

The reentrancy attack is the most famous smart contract vulnerability. It drained 60 million dollars from The DAO in 2016. It still claims victims today. Understanding reentrancy is fundamental to smart contract security.

How Reentrancy Works

When your contract sends ETH to another address, if that address is a contract, it can execute code. That code can call back into your contract before the first call finishes. If your contract has not updated its state, the attacker can exploit this.

Here is a vulnerable withdrawal function:

```
contract VulnerableBank {  
    mapping(address => uint256) public balances;  
  
    function deposit() external payable {  
        balances[msg.sender] += msg.value;  
    }  
}
```

```

function withdraw() external {
    uint256 balance = balances[msg.sender];
    require(balance > 0, "No balance");

    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Transfer failed");

    balances[msg.sender] = 0;
}
}

```

The problem is the order of operations. The contract sends ETH before updating the balance. An attacker deploys this contract:

```

contract Attacker {
    VulnerableBank public bank;
    uint256 public attackCount;

    constructor(address _bank) {
        bank = VulnerableBank(_bank);
    }

    function attack() external payable {
        require(msg.value >= 1 ether, "Need ETH");
        bank.deposit{value: 1 ether}();
        bank.withdraw();
    }

    receive() external payable {
        if (address(bank).balance >= 1 ether && attackCount < 10) {
            attackCount++;
            bank.withdraw();
        }
    }
}

```

The attack flow:

1. Attacker deposits 1 ETH

2. Attacker calls withdraw()
3. Bank checks balance: 1 ETH. Valid.
4. Bank sends 1 ETH to attacker contract
5. Attacker receive() is triggered
6. Before balance is set to 0, attacker calls withdraw() again
7. Bank checks balance: still 1 ETH (not updated yet)
8. Bank sends another 1 ETH
9. This repeats until bank is drained or gas runs out
10. Finally all withdraw calls complete
11. Balance is set to 0 (too late)

The attacker deposited 1 ETH but withdrew many ETH.

The Checks-Effects-Interactions Pattern

The defense is simple: update state before external calls. This is called Checks-Effects-Interactions:

```
contract SecureBank {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 balance = balances[msg.sender];
        require(balance > 0, "No balance");

        balances[msg.sender] = 0;

        (bool success, ) = msg.sender.call{value: balance}("");
        require(success, "Transfer failed");
    }
}
```

Now even if the attacker reenters, their balance is already 0. The second withdraw fails the require check.

The pattern:

1. Checks: Validate all conditions
2. Effects: Update all state
3. Interactions: Make external calls

Never make external calls before updating state. Never.

Reentrancy Guard

For extra protection, use a reentrancy guard. This mutex prevents any reentrant calls:

```
abstract contract ReentrancyGuard {
    uint256 private constant NOT_ENTERED = 1;
    uint256 private constant ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = NOT_ENTERED;
    }

    modifier nonReentrant() {
        require(_status != ENTERED, "ReentrancyGuard: reentrant call");
        _status = ENTERED;
        _;
        _status = NOT_ENTERED;
    }
}

contract SecureBankWithGuard is ReentrancyGuard {
    mapping(address => uint256) public balances;

    function withdraw() external nonReentrant {
        uint256 balance = balances[msg.sender];
        require(balance > 0, "No balance");

        balances[msg.sender] = 0;

        (bool success, ) = msg.sender.call{value: balance}("");
```

```

require(success, "Transfer failed");
}
}

```

Cross-Function Reentrancy

Reentrancy can occur across multiple functions. Consider:

```

contract VulnerableToken {
    mapping(address => uint256) public balances;
    mapping(address => mapping(address => uint256)) public
allowances;

```

```

function transfer(address to, uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient");
    balances[msg.sender] -= amount;
    balances[to] += amount;
}

```

```

function withdrawAll() external {
    uint256 balance = balances[msg.sender];
    require(balance > 0, "No balance");

```

```

(bool success, ) = msg.sender.call{value: balance}("");
require(success, "Transfer failed");

```

```

    balances[msg.sender] = 0;
}
}

```

An attacker can:

1. Call withdrawAll()
2. In receive(), call transfer() to move tokens before balance zeroed
3. Transfer preserves the tokens that should have been burned

The attacker keeps both the ETH and the tokens.

Cross-Contract Reentrancy

When multiple contracts share state or interact, reentrancy can span

contracts:

```
contract TokenVault {
  IERC20 public token;
  mapping(address => uint256) public deposits;

  function deposit(uint256 amount) external {
    token.transferFrom(msg.sender, address(this), amount);
    deposits[msg.sender] += amount;
  }

  function withdraw(uint256 amount) external {
    require(deposits[msg.sender] >= amount, "Insufficient");

    token.transfer(msg.sender, amount);

    deposits[msg.sender] -= amount;
  }
}
```

If the token is malicious or has hooks (like ERC777), the transfer can trigger code execution that calls back into the vault or a related contract.

SECTION 3: INTEGER OVERFLOW AND UNDERFLOW

Before Solidity 0.8, arithmetic operations could overflow or underflow silently. Adding 1 to the maximum uint256 gave 0. Subtracting 1 from 0 gave the maximum uint256. Attackers exploited this.

The Classic Overflow

```
contract VulnerableToken {
  mapping(address => uint256) public balances;

  function transfer(address to, uint256 amount) external {
    require(balances[msg.sender] - amount >= 0, "Insufficient");
    balances[msg.sender] -= amount;
  }
}
```

```
balances[to] += amount;
}
}
```

The require statement always passes. In Solidity pre-0.8, `balances[msg.sender] - amount` underflows to a huge number if amount exceeds balance. That huge number is always greater than or equal to 0.

An attacker with 0 balance could transfer any amount. The subtraction underflows, wrapping around to near-maximum `uint256`. The attacker now has effectively infinite tokens.

Multiplication Overflow

```
contract VulnerableAuction {
  function calculateTotalPrice(uint256 quantity, uint256
pricePerUnit)
  external
  pure
  returns (uint256)
  {
    return quantity * pricePerUnit;
  }
}
```

If quantity and pricePerUnit are both large, the multiplication overflows. The result wraps around to a small number. An attacker could buy millions of items for nearly nothing.

Solidity 0.8+ Protection

Solidity 0.8 added automatic overflow checks. Operations now revert on overflow instead of wrapping:

```
contract SafeToken {
  mapping(address => uint256) public balances;

  function transfer(address to, uint256 amount) external {
    balances[msg.sender] -= amount;
```



```
balances[to] += amount;
}
}
```

In Solidity 0.8+, the subtraction reverts if amount exceeds balance. No explicit check needed.

However, you can opt out of these checks using unchecked blocks:

```
function unsafeDecrement(uint256 x) external pure returns
(uint256) {
    unchecked {
        return x - 1;
    }
}
```

Use unchecked only when you have mathematically proven overflow is impossible. Gas optimization is not worth security risk.

Edge Cases Still Matter

Even with 0.8+ protection, you need to consider edge cases:

```
contract StillVulnerable {
    function divideReward(uint256 totalReward, uint256 participants)
    external
    pure
    returns (uint256)
    {
        return totalReward / participants;
    }
}
```

Division by zero reverts, but what about division that rounds down? If totalReward is 99 and participants is 100, each participant gets 0. The remaining 99 is stuck forever.

SECTION 4: ACCESS CONTROL VULNERABILITIES

Missing or incorrect access control lets unauthorized users call sensitive functions.

Missing Access Control

```
contract VulnerableVault {  
    address public owner;  
  
    constructor() {  
        owner = msg.sender;  
    }  
  
    function withdrawAll(address to) external {  
        payable(to).transfer(address(this).balance);  
    }  
}
```

Anyone can call withdrawAll and drain the vault. The function should check msg.sender.

Incorrect Access Control

```
contract StillVulnerable {  
    address public owner;  
  
    constructor() {  
        owner = msg.sender;  
    }  
  
    modifier onlyOwner() {  
        require(tx.origin == owner, "Not owner");  
        _;  
    }  
  
    function withdrawAll(address to) external onlyOwner {  
        payable(to).transfer(address(this).balance);  
    }  
}
```

This uses tx.origin instead of msg.sender. An attacker can create a

contract that tricks the owner into calling it. That contract then calls `withdrawAll`. `tx.origin` is still the owner, so the check passes.

```
contract AttackContract {
    VulnerableVault public vault;

    constructor(address _vault) {
        vault = VulnerableVault(_vault);
    }

    function attack() external {
        vault.withdrawAll(address(this));
    }

    receive() external payable {}
}
```

The attacker tricks the vault owner into calling `attack()` (maybe through phishing). The call chain is:

Owner EOA -> AttackContract -> Vault

`tx.origin` = Owner EOA (valid owner)
`msg.sender` = AttackContract (not owner)

The check passes because `tx.origin` is the owner.

Always use `msg.sender` for access control:

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}
```

Unprotected Initializers

Upgradeable contracts use initializers instead of constructors:

```
contract VulnerableUpgradeable {
    address public owner;
```

```
bool public initialized;
```

```
function initialize() external {  
    require(!initialized, "Already initialized");  
    owner = msg.sender;  
    initialized = true;  
}  
}
```

If someone calls initialize before the legitimate owner, they become the owner. This happens when:

1. Contract is deployed but not initialized in same transaction
2. Implementation contract is deployed without initialization

Always initialize in the deployment transaction or protect the initializer.

SECTION 5: FRONT-RUNNING AND MEV

Transaction ordering manipulation is a fundamental blockchain vulnerability. Miners (or validators) choose transaction order. Attackers exploit this.

The Sandwich Attack

User wants to buy tokens on Uniswap. They submit a transaction to buy 10 ETH worth of tokens. An attacker sees this in the mempool:

1. Attacker front-runs: Buys tokens first, raising the price
2. User transaction executes: Buys at higher price
3. Attacker back-runs: Sells tokens at the inflated price

The attacker profits from the price movement caused by the user.

Vulnerable Pattern:

```
contract VulnerableSwap {  
    function swap(address tokenIn, address tokenOut, uint256
```

```

amountIn)
external
returns (uint256 amountOut)
{
    amountOut = calculateOutput(tokenIn, tokenOut, amountIn);

    IERC20(tokenIn).transferFrom(msg.sender, address(this),
amountIn);
    IERC20(tokenOut).transfer(msg.sender, amountOut);
}
}

```

No slippage protection. The user accepts any output amount.

Protected Pattern:

```

contract ProtectedSwap {
    function swap(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        uint256 deadline
    )
    external
    returns (uint256 amountOut)
    {
        require(block.timestamp <= deadline, "Expired");

        amountOut = calculateOutput(tokenIn, tokenOut, amountIn);
        require(amountOut >= minAmountOut, "Slippage exceeded");

        IERC20(tokenIn).transferFrom(msg.sender, address(this),
amountIn);
        IERC20(tokenOut).transfer(msg.sender, amountOut);
    }
}

```

The user specifies minimum acceptable output. If sandwich attack moves the price too much, the transaction reverts. The deadline

prevents old transactions from executing at bad prices.

Commit-Reveal Pattern

For actions where knowing the value matters (auctions, voting), use commit-reveal:

```
contract SecureAuction {
    mapping(address => bytes32) public commitments;
    mapping(address => uint256) public bids;

    uint256 public commitDeadline;
    uint256 public revealDeadline;

    function commit(bytes32 commitment) external {
        require(block.timestamp < commitDeadline, "Commit phase ended");
        commitments[msg.sender] = commitment;
    }

    function reveal(uint256 bid, bytes32 secret) external payable {
        require(block.timestamp >= commitDeadline, "Commit phase active");
        require(block.timestamp < revealDeadline, "Reveal phase ended");
        require(msg.value == bid, "Bid mismatch");

        bytes32 commitment = keccak256(abi.encodePacked(bid, secret, msg.sender));
        require(commitment == commitments[msg.sender], "Invalid reveal");

        bids[msg.sender] = bid;
        commitments[msg.sender] = bytes32(0);
    }
}
```

In the commit phase, users submit hashed bids. No one knows the actual values. In the reveal phase, users reveal their bids. Front-running is useless because the commitment already locked in the value.

SECTION 6: ORACLE MANIPULATION

Smart contracts need external data. Price feeds. Random numbers. Real-world events. Oracles provide this data. If the oracle can be manipulated, the contract can be exploited.

Spot Price Manipulation

```
contract VulnerableLending {
    IUniswapPair public pair;
    IERC20 public collateralToken;
    IERC20 public loanToken;

    function getPrice() public view returns (uint256) {
        (uint112 reserve0, uint112 reserve1, ) = pair.getReserves();
        return (uint256(reserve1) * 1e18) / uint256(reserve0);
    }

    function borrow(uint256 collateralAmount) external returns
    (uint256) {
        uint256 price = getPrice();
        uint256 loanAmount = (collateralAmount * price) / 1e18;

        collateralToken.transferFrom(msg.sender, address(this),
        collateralAmount);
        loanToken.transfer(msg.sender, loanAmount);

        return loanAmount;
    }
}
```

The price comes directly from Uniswap reserves. An attacker can:

1. Flash loan huge amount of token0
2. Swap it for token1, manipulating the ratio
3. Call borrow() with the manipulated price
4. Get more loan than collateral is worth
5. Repay flash loan

6. Keep the excess loan tokens

The attack happens in one transaction. The attacker profits from the price manipulation.

Time-Weighted Average Price (TWAP)

TWAP oracles resist single-block manipulation:

```
contract TWAPOracle {
    IUniswapPair public pair;

    uint256 public price0CumulativeLast;
    uint256 public price1CumulativeLast;
    uint256 public blockTimestampLast;

    uint256 public price0Average;
    uint256 public price1Average;

    uint256 public constant PERIOD = 1 hours;

    function update() external {
        (uint256 price0Cumulative, uint256 price1Cumulative, uint256
        blockTimestamp) =
        currentCumulativePrices();

        uint256 timeElapsed = blockTimestamp - blockTimestampLast;

        if (timeElapsed >= PERIOD) {
            price0Average = (price0Cumulative - price0CumulativeLast) /
            timeElapsed;
            price1Average = (price1Cumulative - price1CumulativeLast) /
            timeElapsed;

            price0CumulativeLast = price0Cumulative;
            price1CumulativeLast = price1Cumulative;
            blockTimestampLast = blockTimestamp;
        }
    }
}
```


TWAP averages prices over time. Manipulating it requires sustained price manipulation across many blocks, which is expensive and risky for attackers.

Multiple Oracle Sources

Use multiple independent oracles and compare:

```
contract MultiOracleLending {
  IChainlinkOracle public chainlinkOracle;
  ITWAPOracle public twapOracle;

  uint256 public constant MAX_DEVIATION = 500;

  function getPrice() public view returns (uint256) {
    uint256 chainlinkPrice = chainlinkOracle.latestAnswer();
    uint256 twapPrice = twapOracle.consult();

    uint256 deviation;
    if (chainlinkPrice > twapPrice) {
      deviation = ((chainlinkPrice - twapPrice) * 10000) /
chainlinkPrice;
    } else {
      deviation = ((twapPrice - chainlinkPrice) * 10000) / twapPrice;
    }

    require(deviation <= MAX_DEVIATION, "Price deviation too
high");

    return (chainlinkPrice + twapPrice) / 2;
  }
}
```

SECTION 7: SIGNATURE VULNERABILITIES

Cryptographic signatures prove identity and authorize actions. Vulnerabilities in signature handling enable serious attacks.

Signature Replay

```
contract VulnerablePermit {
    mapping(address => uint256) public balances;

    function permitTransfer(
        address from,
        address to,
        uint256 amount,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        bytes32 hash = keccak256(abi.encodePacked(from, to, amount));
        address signer = ecrecover(hash, v, r, s);

        require(signer == from, "Invalid signature");

        balances[from] -= amount;
        balances[to] += amount;
    }
}
```

The same signature can be used multiple times. An attacker captures a valid signature and replays it to drain the account.

Nonce Protection:

```
contract SecurePermit {
    mapping(address => uint256) public balances;
    mapping(address => uint256) public nonces;

    function permitTransfer(
        address from,
        address to,
        uint256 amount,
        uint256 nonce,
        uint256 deadline,
        uint8 v,
        bytes32 r,
```

```

bytes32 s
) external {
require(block.timestamp <= deadline, "Expired");
require(nonce == nonces[from], "Invalid nonce");

bytes32 hash = keccak256(abi.encodePacked(
from, to, amount, nonce, deadline, address(this), block.chainid
));
bytes32 ethSignedHash = keccak256(abi.encodePacked(
"\x19Ethereum Signed Message:\n32", hash
));

address signer = ecrecover(ethSignedHash, v, r, s);
require(signer == from, "Invalid signature");

nonces[from] ++;

balances[from] -= amount;
balances[to] += amount;
}
}

```

The nonce increments after each use. The same signature cannot be used twice. The chain ID prevents cross-chain replay. The contract address prevents replay on different contracts.

Signature Malleability

ECDSA signatures have malleability. For a valid signature (v, r, s), there exists another valid signature (v', r, s') for the same message. This can break contracts that use signatures as unique identifiers:

```

contract VulnerableVoting {
mapping(bytes32 => bool) public usedSignatures;
mapping(address => uint256) public votes;

function vote(
address voter,
uint256 proposal,
uint8 v,

```

```

bytes32 r,
bytes32 s
) external {
bytes32 sigHash = keccak256(abi.encodePacked(v, r, s));
require(!usedSignatures[sigHash], "Signature used");

bytes32 messageHash = keccak256(abi.encodePacked(voter,
proposal));
address signer = ecrecover(messageHash, v, r, s);
require(signer == voter, "Invalid signature");

usedSignatures[sigHash] = true;
votes[proposal] ++;
}
}

```

An attacker can take a valid signature and compute the malleable counterpart. Both hash differently but recover the same signer. One voter can vote twice.

Fix by canonicalizing s:

```

function vote(
address voter,
uint256 proposal,
uint8 v,
bytes32 r,
bytes32 s
) external {
require(uint256(s) <=
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F4668
"Invalid s");

bytes32 messageHash = keccak256(abi.encodePacked(voter,
proposal));
address signer = ecrecover(messageHash, v, r, s);
require(signer == voter, "Invalid signature");
require(!hasVoted[voter][proposal], "Already voted");

hasVoted[voter][proposal] = true;

```

```
votes[proposal] + +;  
}
```

Even better, track who voted rather than which signatures were used.

SECTION 8: DENIAL OF SERVICE

DoS attacks prevent legitimate users from using the contract.

Unbounded Loops

```
contract VulnerableAirdrop {  
    address[] public recipients;  
  
    function addRecipient(address recipient) external {  
        recipients.push(recipient);  
    }  
  
    function distribute(IERC20 token, uint256 amountPerRecipient)  
    external {  
        for (uint256 i = 0; i < recipients.length; i + +) {  
            token.transfer(recipients[i], amountPerRecipient);  
        }  
    }  
}
```

If recipients array grows large enough, distribute() exceeds block gas limit. No one can execute it. Funds are stuck.

Fix with pull pattern:

```
contract PullAirdrop {  
    mapping(address => uint256) public claimable;  
  
    function setClaimable(address[] calldata recipients, uint256  
    amount) external {  
        for (uint256 i = 0; i < recipients.length; i + +) {  
            claimable[recipients[i]] = amount;  
        }  
    }  
}
```

```

}
}

function claim(IERC20 token) external {
    uint256 amount = claimable[msg.sender];
    require(amount > 0, "Nothing to claim");

    claimable[msg.sender] = 0;
    token.transfer(msg.sender, amount);
}
}

```

Each user claims individually. No unbounded loop.

External Call Failure

```

contract VulnerableRefund {
    mapping(address => uint256) public deposits;
    address[] public depositors;

    function deposit() external payable {
        if (deposits[msg.sender] == 0) {
            depositors.push(msg.sender);
        }
        deposits[msg.sender] += msg.value;
    }

    function refundAll() external {
        for (uint256 i = 0; i < depositors.length; i++) {
            address depositor = depositors[i];
            uint256 amount = deposits[depositor];
            deposits[depositor] = 0;

            (bool success, ) = depositor.call{value: amount}("");
            require(success, "Transfer failed");
        }
    }
}

```

An attacker deploys a contract that reverts in receive(). When

refundAll() tries to send to that contract, it fails. The require reverts everything. No one gets refunded.

Fix by continuing on failure:

```
function refundAll() external {
  for (uint256 i = 0; i < depositors.length; i++) {
    address depositor = depositors[i];
    uint256 amount = deposits[depositor];

    if (amount > 0) {
      deposits[depositor] = 0;

      (bool success, ) = depositor.call{value: amount}("");
      if (!success) {
        deposits[depositor] = amount;
      }
    }
  }
}
```

Or use the pull pattern where users withdraw themselves.

SECTION 9: LOGIC BUGS

Logic bugs are flaws in business logic that do not fit neat categories. They require understanding what the contract should do and finding where it fails.

Wrong Comparison

```
contract VulnerableVesting {
  uint256 public vestingEnd;
  mapping(address => uint256) public vested;

  function claim() external {
    require(block.timestamp > vestingEnd, "Still vesting");

    uint256 amount = vested[msg.sender];
```

```
vested[msg.sender] = 0;
```

```
payable(msg.sender).transfer(amount);  
}  
}
```

Using `>` instead of `>=` means users cannot claim at exactly `vestingEnd`. Minor but shows how easy it is to get comparisons wrong.

More severe example:

```
contract VulnerableLoan {  
    function isHealthy(uint256 collateral, uint256 debt, uint256 price)  
    public  
    pure  
    returns (bool)  
    {  
        return collateral * price > debt * 15 / 10;  
    }  
}
```

Integer division truncates. `debt * 15 / 10` loses precision. For `debt = 1`, the result is 1, not 1.5. Small debts can appear healthier than they are.

State Inconsistency

```
contract VulnerableStaking {  
    uint256 public totalStaked;  
    mapping(address => uint256) public staked;  
  
    function stake(uint256 amount) external {  
        IERC20(token).transferFrom(msg.sender, address(this), amount);  
        staked[msg.sender] += amount;  
        totalStaked += amount;  
    }  
  
    function unstake(uint256 amount) external {  
        require(staked[msg.sender] >= amount, "Insufficient stake");
```



```
staked[msg.sender] -= amount;  
IERC20(token).transfer(msg.sender, amount);  
}  
}
```

The unstake function does not decrease totalStaked. Over time, totalStaked becomes wrong. If rewards depend on totalStaked, they calculate incorrectly.

SECTION 10: BUILDING SECURE CONTRACTS

Security Checklist

Before deploying any contract:

Access Control:

- All sensitive functions have proper access control
- Access control uses msg.sender not tx.origin
- Initializers can only be called once
- Default visibility is appropriate

Arithmetic:

- Using Solidity 0.8+ or SafeMath
- Division by zero is handled
- Rounding errors are acceptable
- Extreme values are considered

External Interactions:

- Checks-Effects-Interactions pattern followed
- Reentrancy guards on sensitive functions
- Low-level calls check return values
- Arbitrary external calls are avoided

Oracle and Price:

- Using TWAP or multiple oracle sources
- Slippage protection on swaps
- Manipulation resistance considered

Signatures:

- Replay protection with nonces
- Chain ID and contract address in hash
- Signature malleability handled

Denial of Service:

- No unbounded loops
- Pull pattern for distributions
- Individual call failures do not block others

Business Logic:

- Edge cases tested
- State consistency maintained
- Comparisons are correct

Comprehensive Security Example

contract SecureVault is ReentrancyGuard, Ownable, Pausable {
using SafeERC20 for IERC20;

IERC20 public immutable token;

mapping(address => uint256) public deposits;

mapping(address => uint256) public nonces;

uint256 public totalDeposits;

uint256 public withdrawalDelay;

mapping(address => uint256) public pendingWithdrawals;

mapping(address => uint256) public withdrawalTimestamps;

event Deposited(address indexed user, uint256 amount);

event WithdrawalRequested(address indexed user, uint256
amount);

event Withdrawn(address indexed user, uint256 amount);

constructor(address _token, uint256 _withdrawalDelay) {

require(_token != address(0), "Invalid token");

token = IERC20(_token);

withdrawalDelay = _withdrawalDelay;

}

```
function deposit(uint256 amount) external nonReentrant
whenNotPaused {
    require(amount > 0, "Zero amount");

    deposits[msg.sender] += amount;
    totalDeposits += amount;

    token.safeTransferFrom(msg.sender, address(this), amount);

    emit Deposited(msg.sender, amount);
}
```

```
function requestWithdrawal(uint256 amount) external
nonReentrant whenNotPaused {
    require(amount > 0, "Zero amount");
    require(deposits[msg.sender] >= amount, "Insufficient deposit");
    require(pendingWithdrawals[msg.sender] == 0, "Pending
withdrawal exists");
```

```
    deposits[msg.sender] -= amount;
    totalDeposits -= amount;
```

```
    pendingWithdrawals[msg.sender] = amount;
    withdrawalTimestamps[msg.sender] = block.timestamp;
```

```
    emit WithdrawalRequested(msg.sender, amount);
}
```

```
function completeWithdrawal() external nonReentrant {
    uint256 amount = pendingWithdrawals[msg.sender];
    require(amount > 0, "No pending withdrawal");
```

```
    uint256 timestamp = withdrawalTimestamps[msg.sender];
    require(
        block.timestamp >= timestamp + withdrawalDelay,
        "Withdrawal delay not met"
    );
```

```
    pendingWithdrawals[msg.sender] = 0;
```

```

withdrawalTimestamps[msg.sender] = 0;

token.safeTransfer(msg.sender, amount);

emit Withdrawn(msg.sender, amount);
}

function cancelWithdrawal() external nonReentrant {
    uint256 amount = pendingWithdrawals[msg.sender];
    require(amount > 0, "No pending withdrawal");

    pendingWithdrawals[msg.sender] = 0;
    withdrawalTimestamps[msg.sender] = 0;

    deposits[msg.sender] += amount;
    totalDeposits += amount;
}

function emergencyWithdraw() external onlyOwner {
    uint256 balance = token.balanceOf(address(this));
    token.safeTransfer(owner(), balance);
}

function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
}

```

This vault implements:

1. ReentrancyGuard on all state-changing functions
2. Proper access control with Ownable
3. Pausable for emergencies
4. Withdrawal delay to prevent flash loan attacks
5. Checks-Effects-Interactions pattern
6. SafeERC20 for token transfers

7. State consistency between deposits and totalDeposits
8. Events for transparency
9. Input validation
10. Immutable token to prevent address changes

This chapter covered the major vulnerability categories.

Understanding these attacks is the foundation of smart contract security. In the next chapter, we will examine how to systematically find these vulnerabilities through security auditing.

BOOK 7: SECURITY AND AUDITING

Chapter 2: Security Auditing Process

Auditing smart contracts is systematic work. You cannot randomly look at code hoping to find bugs. You need methodology. Process. Checklists. Tools. This chapter teaches you how professional auditors find vulnerabilities that developers miss.

SECTION 1: THE AUDITING MINDSET

What Auditors Do

Auditors think adversarially. Every line of code is a potential attack vector. Every assumption is a potential vulnerability. Every external interaction is an opportunity for exploitation.

The goal is not to prove code is secure. The goal is to try everything possible to break it. If you cannot break it after exhaustive effort, then maybe it is secure.

Auditors ask:

What happens if this input is zero?

What happens if this input is maximum uint256?

What happens if this function is called before that function?

What happens if this function is called by a contract instead of an EOA?

What happens if the external contract behaves maliciously?

What happens if this transaction is front-run?

What happens if many users call this simultaneously?

Types of Audits

Manual Review:

Human auditors read code line by line. They understand business logic. They catch issues tools miss. They reason about complex attack scenarios. This is essential but slow and expensive.

Automated Analysis:

Tools scan code for patterns. They find common vulnerabilities quickly. They do not understand business logic. They produce false positives. They miss novel attacks. This is fast but incomplete.

Formal Verification:

Mathematical proofs that code satisfies specifications. Provides highest assurance. Requires significant expertise. Limited to properties that can be formally specified. Time-consuming and expensive.

The best audits combine all three. Automated tools find obvious issues. Manual review catches logic bugs. Formal verification proves critical properties.

SECTION 2: AUDIT PREPARATION

Understanding the Project

Before reading code, understand the project:

What does this protocol do?

What are the main user flows?

What assets does it handle?

What external protocols does it integrate?

What are the trust assumptions?

Who are the privileged roles?

What economic incentives exist?

Read the documentation. Read the whitepaper. Understand the

business logic before the code logic.

Setting Up the Environment

Clone the repository. Install dependencies. Make sure tests pass:

```
git clone https://github.com/project/contracts
cd contracts
npm install
npx hardhat compile
npx hardhat test
```

Set up your analysis environment:

```
from dataclasses import dataclass, field
from typing import List, Dict, Optional, Set
from enum import Enum
from pathlib import Path
import json
```

```
class Severity(Enum):
    CRITICAL = "critical"
    HIGH = "high"
    MEDIUM = "medium"
    LOW = "low"
    INFORMATIONAL = "informational"
```

```
class IssueStatus(Enum):
    OPEN = "open"
    CONFIRMED = "confirmed"
    DISPUTED = "disputed"
    FIXED = "fixed"
    ACKNOWLEDGED = "acknowledged"
```

```
@dataclass
class Finding:
    title: str
```

```
severity: Severity
description: str
location: str
impact: str
recommendation: str
status: IssueStatus = IssueStatus.OPEN
poc: Optional[str] = None
references: List[str] = field(default_factory = list)
```

```
@dataclass
class AuditProject:
    name: str
    repository: str
    commit_hash: str
    scope: List[str]
    out_of_scope: List[str]
    documentation: List[str]
    findings: List[Finding] = field(default_factory = list)
```

```
def add_finding(self, finding: Finding) -> None:
    self.findings.append(finding)
```

```
def findings_by_severity(self, severity: Severity) -> List[Finding]:
    return [f for f in self.findings if f.severity == severity]
```

```
def summary(self) -> Dict[str, int]:
    return {
        "critical": len(self.findings_by_severity(Severity.CRITICAL)),
        "high": len(self.findings_by_severity(Severity.HIGH)),
        "medium": len(self.findings_by_severity(Severity.MEDIUM)),
        "low": len(self.findings_by_severity(Severity.LOW)),
        "informational":
            len(self.findings_by_severity(Severity.INFORMATIONAL))
    }
```

```
class AuditTracker:
```

```
    def __init__(self, project: AuditProject):
```



```

self.project = project
self.notes: Dict[str, List[str]] = {}
self.reviewed_functions: Set[str] = set()
self.attack_vectors: List[str] = []

def add_note(self, location: str, note: str) -> None:
    if location not in self.notes:
        self.notes[location] = []
    self.notes[location].append(note)

def mark_reviewed(self, function_signature: str) -> None:
    self.reviewed_functions.add(function_signature)

def add_attack_vector(self, vector: str) -> None:
    self.attack_vectors.append(vector)

def record_finding(
    self,
    title: str,
    severity: str,
    description: str,
    location: str,
    impact: str,
    recommendation: str,
    poc: Optional[str] = None
) -> Finding:

    finding = Finding(
        title=title,
        severity=Severity(severity.lower()),
        description=description,
        location=location,
        impact=impact,
        recommendation=recommendation,
        poc=poc
    )

    self.project.add_finding(finding)
    return finding

```

```
def export_report(self, output_path: str) -> None:
    report = {
        "project": self.project.name,
        "repository": self.project.repository,
        "commit": self.project.commit_hash,
        "scope": self.project.scope,
        "summary": self.project.summary(),
        "findings": []
    }
```

```
for finding in self.project.findings:
    report["findings"].append({
        "title": finding.title,
        "severity": finding.severity.value,
        "description": finding.description,
        "location": finding.location,
        "impact": finding.impact,
        "recommendation": finding.recommendation,
        "status": finding.status.value,
        "poc": finding.poc
    })
```

```
with open(output_path, "w") as f:
    json.dump(report, f, indent=2)
```

Scoping

Define what is in scope:

```
project = AuditProject(
    name="DeFi Protocol V2",
    repository="https://github.com/project/v2",
    commit_hash="abc123def456",
    scope=[
        "contracts/core/Vault.sol",
        "contracts/core/Strategy.sol",
        "contracts/core/Token.sol",
        "contracts/periphery/Router.sol"
    ],
```

```
out_of_scope = [
    "contracts/mocks/",
    "contracts/test/",
    "node_modules/"
],
documentation = [
    "docs/architecture.md",
    "docs/specification.pdf"
]
)
```

```
tracker = AuditTracker(project)
```

SECTION 3: CODE REVIEW METHODOLOGY

First Pass: High-Level Understanding

Read through all contracts quickly. Do not look for bugs yet.
Understand:

1. Contract structure and inheritance
2. State variables and their purposes
3. Main functions and their flows
4. External calls and dependencies
5. Access control patterns
6. Event emissions

Create a mental map of how contracts interact:

```
from typing import Dict, List, Set, Tuple
from dataclasses import dataclass, field
```

```
@dataclass
class ContractInfo:
    name: str
    file_path: str
    inherits_from: List[str] = field(default_factory=list)
    state_variables: List[str] = field(default_factory=list)
```

```
functions: List[str] = field(default_factory = list)
external_calls: List[str] = field(default_factory = list)
events: List[str] = field(default_factory = list)
modifiers: List[str] = field(default_factory = list)
```

```
class CodebaseAnalyzer:
```

```
def __init__(self):
self.contracts: Dict[str, ContractInfo] = {}
self.call_graph: Dict[str, Set[str]] = {}
self.inheritance_graph: Dict[str, Set[str]] = {}
```

```
def add_contract(self, contract: ContractInfo) -> None:
self.contracts[contract.name] = contract
```

```
for parent in contract.inherits_from:
if parent not in self.inheritance_graph:
self.inheritance_graph[parent] = set()
self.inheritance_graph[parent].add(contract.name)
```

```
def build_call_graph(self) -> None:
for contract_name, contract in self.contracts.items():
self.call_graph[contract_name] = set()
```

```
for ext_call in contract.external_calls:
target = ext_call.split(".")[0]
if target in self.contracts:
self.call_graph[contract_name].add(target)
```

```
def find_entry_points(self) -> List[Tuple[str, str]]:
entry_points = []
```

```
for contract_name, contract in self.contracts.items():
for func in contract.functions:
if "external" in func or "public" in func:
entry_points.append((contract_name, func))
```

```
return entry_points
```

```

def get_attack_surface(self, contract_name: str) -> Dict[str,
List[str]]:
    contract = self.contracts.get(contract_name)
    if not contract:
        return {}

    return {
        "external_functions": [f for f in contract.functions if "external" in f],
        "public_functions": [f for f in contract.functions if "public" in f],
        "payable_functions": [f for f in contract.functions if "payable" in f],
        "external_calls": contract.external_calls,
        "state_modifying": [f for f in contract.functions if "view" not in f and
        "pure" not in f]
    }

```

Second Pass: Function-by-Function Review

Go through each function systematically. For every function ask:

Input Validation:

- Are all inputs validated?
- What happens with zero values?
- What happens with maximum values?
- Can the caller be a contract?

State Changes:

- What state is modified?
- Is state updated before external calls?
- Is state consistent after the function?
- Can state be manipulated by reentrancy?

External Calls:

- What external contracts are called?
- Can they be malicious?
- Are return values checked?
- Can they reenter?

Access Control:

- Who can call this function?

- Is the access control correct?
- Can it be bypassed?

Economic Logic:

- Are calculations correct?
- Is there rounding error exploitation?
- Can prices be manipulated?
- Are there flash loan concerns?

class FunctionAnalyzer:

```
def __init__(self, tracker: AuditTracker):
    self.tracker = tracker
    self.checklist = [
        self.check_input_validation,
        self.check_access_control,
        self.check_reentrancy,
        self.check_integer_issues,
        self.check_external_calls,
        self.check_state_consistency,
        self.check_economic_logic
    ]
```

```
def analyze_function(
    self,
    contract: str,
    function_name: str,
    code: str
) -> List[str]:
```

```
    concerns = []
    location = f"{contract}::{function_name}"
```

```
    for check in self.checklist:
        result = check(code, location)
        if result:
            concerns.append(result)
```

```
    self.tracker.mark_reviewed(location)
    return concerns
```

```

def check_input_validation(self, code: str, location: str) ->
Optional[str]:
    dangerous_patterns = [
        ("address(0)", "Missing zero address check"),
        ("amount == 0", "Missing zero amount check"),
        ("length", "Array length not validated")
    ]

```

```

    for pattern, concern in dangerous_patterns:
        if pattern not in code.lower():
            external_facing = "external" in code or "public" in code
            if external_facing:
                has_input = "uint" in code or "address" in code
                if has_input:
                    return f"{location}: {concern}"

```

```

    return None

```

```

def check_access_control(self, code: str, location: str) ->
Optional[str]:
    state_modifying = "view" not in code and "pure" not in code
    external_facing = "external" in code or "public" in code

    if state_modifying and external_facing:
        access_patterns = ["onlyOwner", "onlyRole", "require(msg.sender",
                           "require(_msgSender)"]
        has_access_control = any(p in code for p in access_patterns)

```

```

    if not has_access_control:
        return f"{location}: External state-modifying function without
        apparent access control"

```

```

    return None

```

```

def check_reentrancy(self, code: str, location: str) -> Optional[str]:
    has_external_call = ".call{" in code or ".transfer(" in code or
    "safeTransfer" in code

```

```

    if has_external_call:

```

```
has_guard = "nonReentrant" in code
if not has_guard:
    return f"{location}: External call without reentrancy guard"

return None
```

```
def check_integer_issues(self, code: str, location: str) ->
Optional[str]:
    if "unchecked" in code:
        return f"{location}: Uses unchecked arithmetic - verify overflow
safety"

    if "/" in code:
        return f"{location}: Division detected - check for precision loss and
division by zero"

    return None
```

```
def check_external_calls(self, code: str, location: str) ->
Optional[str]:
    if ".call{" in code:
        if "require(success" not in code and "(bool success" not in code:
            return f"{location}: Low-level call without return value check"

    return None
```

```
def check_state_consistency(self, code: str, location: str) ->
Optional[str]:
    state_vars = ["mapping", "uint256 public", "address public", "bool
public"]
    modifies_state = any(v in code for v in state_vars)

    if modifies_state:
        self.tracker.add_note(location, "Verify state consistency after
function execution")

    return None
```

```
def check_economic_logic(self, code: str, location: str) ->
Optional[str]:
```



```

economic_patterns = ["price", "amount", "balance", "reserve",
"liquidity"]
has_economic_logic = any(p in code.lower() for p in
economic_patterns)

if has_economic_logic:
    return f'{location}: Contains economic logic - verify against
manipulation"

return None

```

Third Pass: Cross-Function Analysis

Look at how functions interact:

```
class CrossFunctionAnalyzer:
```

```

    def __init__(self, tracker: AuditTracker):
        self.tracker = tracker
        self.state_modifiers: Dict[str, List[str]] = {}
        self.state_readers: Dict[str, List[str]] = {}

```

```

    def register_state_access(
        self,
        function: str,
        variable: str,
        is_write: bool
    ) -> None:

```

```

        if is_write:
            if variable not in self.state_modifiers:
                self.state_modifiers[variable] = []
                self.state_modifiers[variable].append(function)
            else:
                if variable not in self.state_readers:
                    self.state_readers[variable] = []
                    self.state_readers[variable].append(function)

```

```

    def find_ordering_dependencies(self) -> List[Dict[str, any]]:

```

```
dependencies = []
```

```
for variable in self.state_modifiers:
```

```
writers = self.state_modifiers.get(variable, [])
```

```
readers = self.state_readers.get(variable, [])
```

```
if len(writers) > 1 or (writers and readers):
```

```
dependencies.append({
```

```
"variable": variable,
```

```
"writers": writers,
```

```
"readers": readers,
```

```
"concern": "Potential ordering dependency or race condition"
```

```
})
```

```
return dependencies
```

```
def find_cross_function_reentrancy(self) -> List[str]:
```

```
concerns = []
```

```
for variable, modifiers in self.state_modifiers.items():
```

```
if len(modifiers) > 1:
```

```
concern = f"Variable {variable} modified by multiple functions:  
{modifiers}"
```

```
concern += " - Check for cross-function reentrancy"
```

```
concerns.append(concern)
```

```
self.tracker.add_attack_vector(concern)
```

```
return concerns
```

```
def analyze_invariants(self, invariants: List[str]) -> List[str]:
```

```
violations = []
```

```
for invariant in invariants:
```

```
relevant_modifiers = []
```

```
for var, funcs in self.state_modifiers.items():
```

```
if var in invariant:
```

```
relevant_modifiers.extend(funcs)
```

```
if relevant_modifiers:
```

```
check = f"Invariant '{invariant}' may be affected by:
```

```
{relevant_modifiers}"
violations.append(check)
```

```
return violations
```

SECTION 4: VULNERABILITY PATTERNS

Pattern Recognition

Train yourself to recognize dangerous patterns instantly:

class PatternMatcher:

```
def __init__(self):
    self.patterns = {
        "reentrancy": [
            (r"\.call\{.*value:", "Low-level call with value"),
            (r"\.transfer\(", "Transfer before state update"),
            (r"\.send\(", "Send before state update")
        ],
        "access_control": [
            (r"tx\.origin", "tx.origin used for authentication"),
            (r"onlyOwner", "Centralized owner control"),
            (r"private\s+\w+\s*=", "Private variable accessible via storage slot")
        ],
        "integer": [
            (r"unchecked\s*\{", "Unchecked arithmetic block"),
            (r"\*\s*\d+\s*/", "Multiplication before division"),
            (r"/\s*\d+\s*\*", "Division before multiplication - precision loss")
        ],
        "oracle": [
            (r"getReserves\(\)", "Using spot price from reserves"),
            (r"balanceOf.*price", "Price from balance - manipulable"),
            (r"latestAnswer\(\)", "Chainlink without freshness check")
        ],
        "dos": [
            (r"for\s*\[.*\];.*\s*\w+\s*<\s*\w+\s*\.\length", "Unbounded loop"),
            (r"require.*\.call", "Require on external call"),
```

```
(r"delete\s+\w+\[", "Array element deletion without reorg")
],
"signature": [
(r"ecrecover", "ECDSA recovery - check malleability"),
(r"keccak256.*abi\.encodePacked", "Hash collision risk with
encodePacked"),
(r"v,\s*r,\s*s", "Signature parameters - check replay protection")
]
}
```

```
def scan_code(self, code: str) -> Dict[str, List[Tuple[str, int]]]:
import re
```

```
findings = {}
```

```
lines = code.split("\n")
```

```
for category, patterns in self.patterns.items():
findings[category] = []
```

```
for pattern, description in patterns:
for line_num, line in enumerate(lines, 1):
if re.search(pattern, line):
findings[category].append((description, line_num))
```

```
return {k: v for k, v in findings.items() if v}
```

Known Vulnerability Checklist

```
class VulnerabilityChecklist:
```

```
def __init__(self):
self.checklist = {
"SWC-100": {
"name": "Function Default Visibility",
"check": self.check_default_visibility
},
"SWC-101": {
"name": "Integer Overflow and Underflow",
```

```
"check": self.check_integer_overflow
},
"SWC-104": {
  "name": "Unchecked Call Return Value",
  "check": self.check_unchecked_return
},
"SWC-106": {
  "name": "Unprotected SELFDESTRUCT",
  "check": self.check_selfdestruct
},
"SWC-107": {
  "name": "Reentrancy",
  "check": self.check_reentrancy
},
"SWC-108": {
  "name": "State Variable Default Visibility",
  "check": self.check_state_visibility
},
"SWC-110": {
  "name": "Assert Violation",
  "check": self.check_assert_usage
},
"SWC-111": {
  "name": "Use of Deprecated Functions",
  "check": self.check_deprecated
},
"SWC-113": {
  "name": "DoS with Failed Call",
  "check": self.check_dos_failed_call
},
"SWC-114": {
  "name": "Transaction Order Dependence",
  "check": self.check_front_running
},
"SWC-115": {
  "name": "Authorization through tx.origin",
  "check": self.check_tx_origin
},
"SWC-116": {
  "name": "Block Timestamp Manipulation",
```

```

"check": self.check_timestamp
},
"SWC-120": {
"name": "Weak Sources of Randomness",
"check": self.check_randomness
},
"SWC-124": {
"name": "Write to Arbitrary Storage",
"check": self.check_arbitrary_storage
},
"SWC-128": {
"name": "DoS with Block Gas Limit",
"check": self.check_gas_limit
},
"SWC-131": {
"name": "Presence of Unused Variables",
"check": self.check_unused_vars
},
"SWC-136": {
"name": "Unencrypted Private Data",
"check": self.check_private_data
}
}

```

```

def run_checklist(self, code: str, contract: str) -> List[Dict[str, str]]:
    findings = []

```

```

    for swc_id, check_info in self.checklist.items():
        result = check_info["check"](code)
        if result:
            findings.append({
                "swc": swc_id,
                "name": check_info["name"],
                "contract": contract,
                "details": result
            })

```

```

    return findings

```

```

def check_default_visibility(self, code: str) -> Optional[str]:

```

```

import re
pattern = r"function\s+\w+\s*\([^\)]*\)\s*(?:returns|{})"
matches = re.findall(pattern, code)
if matches:
    return f"Functions without explicit visibility: {len(matches)} found"
return None

```

```

def check_integer_overflow(self, code: str) -> Optional[str]:
    if "pragma solidity" in code:
        version_match = code.find("0.8")
        if version_match == -1 and "SafeMath" not in code:
            return "Solidity < 0.8 without SafeMath"
        return None

```

```

def check_unchecked_return(self, code: str) -> Optional[str]:
    if ".call{" in code or ".call(" in code:
        if "bool success" not in code and "(bool" not in code:
            return "Low-level call without return value check"
        return None

```

```

def check_selfdestruct(self, code: str) -> Optional[str]:
    if "selfdestruct" in code or "suicide" in code:
        return "Contract contains selfdestruct"
    return None

```

```

def check_reentrancy(self, code: str) -> Optional[str]:
    if ".call{value:" in code and "nonReentrant" not in code:
        return "ETH transfer without reentrancy protection"
    return None

```

```

def check_state_visibility(self, code: str) -> Optional[str]:
    import re
    pattern = r"^\s*(uint|int|address|bool|bytes|string|mapping)\s+\w+"
    matches = re.findall(pattern, code, re.MULTILINE)
    if matches:
        return f"State variables without explicit visibility found"
    return None

```

```

def check_assert_usage(self, code: str) -> Optional[str]:

```

```
if "assert(" in code:
    return "Assert used - consider require for input validation"
return None
```

```
def check_deprecated(self, code: str) -> Optional[str]:
    deprecated = ["suicide", "throw", "sha3", "callcode", "var "]
    found = [d for d in deprecated if d in code]
    if found:
        return f'Deprecated functions/keywords: {found}'
    return None
```

```
def check_dos_failed_call(self, code: str) -> Optional[str]:
    if "require" in code and ".call" in code:
        return "Require on external call may cause DoS"
    return None
```

```
def check_front_running(self, code: str) -> Optional[str]:
    sensitive = ["price", "swap", "buy", "sell", "bid", "auction"]
    if any(s in code.lower() for s in sensitive):
        if "deadline" not in code.lower() and "minAmount" not in
code.lower():
            return "Price-sensitive operation without front-running protection"
    return None
```

```
def check_tx_origin(self, code: str) -> Optional[str]:
    if "tx.origin" in code:
        if "require" in code or "if" in code:
            return "tx.origin used - vulnerable to phishing"
    return None
```

```
def check_timestamp(self, code: str) -> Optional[str]:
    if "block.timestamp" in code or "now" in code:
        return "Block timestamp used - miners can manipulate within ~15
seconds"
    return None
```

```
def check_randomness(self, code: str) -> Optional[str]:
    weak_random = ["block.timestamp", "block.difficulty",
"block.number", "blockhash"]
    if any(r in code for r in weak_random):
```



```
if "random" in code.lower() or "lottery" in code.lower():  
    return "Weak randomness source detected"  
return None
```

```
def check_arbitrary_storage(self, code: str) -> Optional[str]:  
    if "delegatecall" in code:  
        return "Delegatecall present - verify storage layout compatibility"  
    return None
```

```
def check_gas_limit(self, code: str) -> Optional[str]:  
    import re  
    if re.search(r"for.*\\.length", code) or re.search(r"while.*true",  
code):  
        return "Potentially unbounded iteration"  
    return None
```

```
def check_unused_vars(self, code: str) -> Optional[str]:  
    return None
```

```
def check_private_data(self, code: str) -> Optional[str]:  
    sensitive = ["password", "secret", "private_key", "apikey"]  
    if any(s in code.lower() for s in sensitive):  
        return "Sensitive data in contract - all storage is publicly readable"  
    return None
```

SECTION 5: TESTING VULNERABILITIES

Writing Proof of Concept

When you find a potential vulnerability, prove it works. A PoC removes doubt and demonstrates severity:

```
from dataclasses import dataclass  
from typing import Optional, List  
from web3 import Web3  
from eth_account import Account
```

```
@dataclass
```

```
class PoCResult:
    success: bool
    description: str
    initial_state: dict
    final_state: dict
    profit: Optional[int] = None
    transactions: List[str] = None
```

```
class ReentrancyPoC:
```

```
    def __init__(self, web3: Web3, target_address: str, attacker_key: str):
        self.w3 = web3
        self.target = target_address
        self.attacker = Account.from_key(attacker_key)
```

```
    def deploy_attacker_contract(self, attack_contract_bytecode: str) ->
    str:
```

```
        tx = {
            "from": self.attacker.address,
            "data": attack_contract_bytecode,
            "gas": 3000000,
            "gasPrice": self.w3.eth.gas_price,
            "nonce": self.w3.eth.get_transaction_count(self.attacker.address)
        }
```

```
        signed = self.attacker.sign_transaction(tx)
        tx_hash = self.w3.eth.send_raw_transaction(signed.rawTransaction)
        receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)
```

```
        return receipt.contractAddress
```

```
    def execute_attack(
        self,
        attacker_contract: str,
        attack_function: str,
        attack_value: int
    ) -> PoCResult:
```

```
        target_balance_before = self.w3.eth.get_balance(self.target)
```

```

attacker_balance_before =
self.w3.eth.get_balance(self.attacker.address)

attack_contract_instance = self.w3.eth.contract(
address=attacker_contract,
abi=self.get_attacker_abi()
)

tx = attack_contract_instance.functions[attack_function]
().build_transaction({
"from": self.attacker.address,
"value": attack_value,
"gas": 500000,
"gasPrice": self.w3.eth.gas_price,
"nonce": self.w3.eth.get_transaction_count(self.attacker.address)
})

signed = self.attacker.sign_transaction(tx)
tx_hash = self.w3.eth.send_raw_transaction(signed.rawTransaction)

try:
receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)
except Exception as e:
return PoCResult(
success=False,
description=f"Attack transaction reverted: {e}",
initial_state={"target": target_balance_before},
final_state={"target": target_balance_before}
)

target_balance_after = self.w3.eth.get_balance(self.target)
attacker_balance_after = self.w3.eth.get_balance(attacker_contract)

drained = target_balance_before - target_balance_after
profit = attacker_balance_after - attack_value

return PoCResult(
success=profit > 0,
description=f"Drained {drained} wei from target, profit {profit} wei",

```

```

initial_state = {
    "target_balance": target_balance_before,
    "attacker_balance": attacker_balance_before
},
final_state = {
    "target_balance": target_balance_after,
    "attacker_balance": attacker_balance_after
},
profit = profit,
transactions = [tx_hash.hex()]
)

```

```

def get_attacker_abi(self) -> list:
    return [
        {
            "inputs": [],
            "name": "attack",
            "outputs": [],
            "stateMutability": "payable",
            "type": "function"
        },
        {
            "inputs": [],
            "name": "withdraw",
            "outputs": [],
            "stateMutability": "nonpayable",
            "type": "function"
        }
    ]

```

```

class FlashLoanPoC:

```

```

    def __init__(self, web3: Web3, lending_pool: str, target: str):
        self.w3 = web3
        self.lending_pool = lending_pool
        self.target = target

    def calculate_attack_profitability(
        self,

```

```

loan_amount: int,
expected_profit: int,
flash_loan_fee: float = 0.0009
) -> dict:

fee = int(loan_amount * flash_loan_fee)
gas_cost = 500000 * self.w3.eth.gas_price
net_profit = expected_profit - fee - gas_cost

return {
"loan_amount": loan_amount,
"flash_loan_fee": fee,
"estimated_gas_cost": gas_cost,
"gross_profit": expected_profit,
"net_profit": net_profit,
"profitable": net_profit > 0
}

```

Fuzz Testing

Fuzz testing feeds random inputs to find edge cases:

```

import random
from typing import Callable, List, Any, Dict
from dataclasses import dataclass

@dataclass
class FuzzResult:
    iterations: int
    crashes: List[Dict[str, Any]]
    unique_crashes: int

class ContractFuzzer:

    def __init__(self, web3: Web3, contract_address: str, abi: list):
        self.w3 = web3
        self.contract = self.w3.eth.contract(address=contract_address,

```

```

abi = abi)
self.crashes: List[Dict[str, Any]] = []

def generate_random_uint256(self) -> int:
    edge_cases = [
        0,
        1,
        2**256 - 1,
        2**255,
        2**128,
        2**64,
        2**32,
        2**16,
        2**8
    ]

    if random.random() < 0.3:
        return random.choice(edge_cases)
    else:
        return random.randint(0, 2**256 - 1)

def generate_random_address(self) -> str:
    edge_cases = [
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000001",
        "0xFFFFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfFfF"
    ]

    if random.random() < 0.2:
        return random.choice(edge_cases)
    else:
        return "0x" + "".join(random.choices("0123456789abcdef", k = 40))

def fuzz_function(
    self,
    function_name: str,
    param_types: List[str],
    iterations: int = 1000,
    value_range: tuple = (0, 10**18)
) -> FuzzResult:

```

```
crashes = []
```

```
for i in range(iterations):
```

```
    params = []
```

```
    for param_type in param_types:
```

```
        if "uint" in param_type:
```

```
            params.append(self.generate_random_uint256())
```

```
        elif "address" in param_type:
```

```
            params.append(self.generate_random_address())
```

```
        elif "bool" in param_type:
```

```
            params.append(random.choice([True, False]))
```

```
        elif "bytes" in param_type:
```

```
            length = random.randint(0, 100)
```

```
            params.append(bytes(random.randint(0, 255) for _ in  
range(length)))
```

```
        value = random.randint(*value_range) if random.random() < 0.3  
        else 0
```

```
    try:
```

```
        func = getattr(self.contract.functions, function_name)
```

```
        result = func(*params).call({"value": value})
```

```
    except Exception as e:
```

```
        crash = {
```

```
            "iteration": i,
```

```
            "function": function_name,
```

```
            "params": params,
```

```
            "value": value,
```

```
            "error": str(e)
```

```
        }
```

```
        crashes.append(crash)
```

```
unique_errors = len(set(c["error"] for c in crashes))
```

```
return FuzzResult(  
    iterations=iterations,  
    crashes=crashes,  
    unique_crashes=unique_errors  
)
```

SECTION 6: AUTOMATED ANALYSIS TOOLS

Static Analysis with Slither

Slither is the most popular static analyzer for Solidity. Run it on every audit:

```
import subprocess
import json
from typing import Dict, List, Optional
from dataclasses import dataclass


@dataclass
class SlitherFinding:
    check: str
    impact: str
    confidence: str
    description: str
    elements: List[Dict]


class SlitherAnalyzer:

    def __init__(self, project_path: str):
        self.project_path = project_path

    def run_analysis(self) -> List[SlitherFinding]:
        result = subprocess.run(
            ["slither", self.project_path, "--json", "-"],
            capture_output=True,
            text=True
        )

        if result.returncode != 0 and not result.stdout:
            raise RuntimeError(f'Slither failed: {result.stderr}')

        output = json.loads(result.stdout)
```



```

findings = []
for detector in output.get("results", {}).get("detectors", []):
    finding = SlitherFinding(
        check=detector.get("check", "unknown"),
        impact=detector.get("impact", "unknown"),
        confidence=detector.get("confidence", "unknown"),
        description=detector.get("description", ""),
        elements=detector.get("elements", [])
    )
    findings.append(finding)

return findings

def run_specific_detector(self, detector: str) -> List[SlitherFinding]:
    result = subprocess.run(
        ["slither", self.project_path, "--detect", detector, "--json", "-"],
        capture_output=True,
        text=True
    )

    if result.returncode != 0 and not result.stdout:
        return []

    output = json.loads(result.stdout)

    findings = []
    for det in output.get("results", {}).get("detectors", []):
        finding = SlitherFinding(
            check=det.get("check", "unknown"),
            impact=det.get("impact", "unknown"),
            confidence=det.get("confidence", "unknown"),
            description=det.get("description", ""),
            elements=det.get("elements", [])
        )
        findings.append(finding)

    return findings

def get_contract_summary(self) -> Dict:

```

```

result = subprocess.run(
["slither", self.project_path, "--print", "contract-summary", "--json",
"-"],
capture_output=True,
text=True
)

```

```

if result.stdout:
return json.loads(result.stdout)
return {}

```

```

def get_function_summary(self) -> Dict:
result = subprocess.run(
["slither", self.project_path, "--print", "function-summary", "--json",
"-"],
capture_output=True,
text=True
)

```

```

if result.stdout:
return json.loads(result.stdout)
return {}

```

Mythril Deep Analysis

Mythril performs symbolic execution to find bugs:

```

class MythrilAnalyzer:

```

```

def __init__(self, contract_path: str):
self.contract_path = contract_path

```

```

def run_analysis(self, execution_timeout: int = 300) -> Dict:
result = subprocess.run(
[
"myth", "analyze",
self.contract_path,
"--execution-timeout", str(execution_timeout),
"-o", "json"

```

```

],
capture_output=True,
text=True
)

```

```

if result.stdout:
    return json.loads(result.stdout)
return {"issues": [], "error": result.stderr}

```

```

def run_specific_modules(self, modules: List[str]) -> Dict:
    module_arg = ",".join(modules)

```

```

    result = subprocess.run(
        [
            "myth", "analyze",
            self.contract_path,
            "--modules", module_arg,
            "-o", "json"
        ],
        capture_output=True,
        text=True
    )

```

```

    if result.stdout:
        return json.loads(result.stdout)
    return {"issues": []}

```

Combined Analysis Pipeline

```

class AuditToolPipeline:

```

```

    def __init__(self, project_path: str, tracker: AuditTracker):
        self.project_path = project_path
        self.tracker = tracker

```

```

    def run_full_pipeline(self) -> Dict[str, List]:
        results = {
            "slither": [],
            "mythril": [],

```

```
"patterns": [],
"checklist": []
}
```

```
slither = SlitherAnalyzer(self.project_path)
try:
    results["slither"] = slither.run_analysis()
    for finding in results["slither"]:
        if finding.impact in ["High", "Medium"]:
            self.tracker.add_note(
                "automated",
                f'Slither {finding.impact}: {finding.check} - {finding.description}'
            )
    except Exception as e:
        self.tracker.add_note("automated", f'Slither failed: {e}')

return results
```

```
def prioritize_findings(self, findings: List[SlitherFinding]) ->
List[SlitherFinding]:
    priority_order = {"High": 0, "Medium": 1, "Low": 2, "Informational":
3}
    confidence_order = {"High": 0, "Medium": 1, "Low": 2}

    return sorted(
        findings,
        key = lambda f: (
            priority_order.get(f.impact, 4),
            confidence_order.get(f.confidence, 3)
        )
    )
```

SECTION 7: DOCUMENTING FINDINGS

Writing Good Finding Reports

Each finding needs clear documentation:

```
class FindingWriter:
```

```
def __init__(self, tracker: AuditTracker):
    self.tracker = tracker
```

```
def create_finding(
    self,
    title: str,
    severity: str,
    contract: str,
    function: str,
    line: int,
    description: str,
    impact: str,
    recommendation: str,
    poc_code: Optional[str] = None
) -> Finding:
```

```
    location = f"{contract}::{function} (Line {line})"
```

```
    finding = self.tracker.record_finding(
        title=title,
        severity=severity,
        description=description,
        location=location,
        impact=impact,
        recommendation=recommendation,
        poc=poc_code
    )
```

```
    return finding
```

```
def generate_finding_markdown(self, finding: Finding) -> str:
    severity_emoji = {
        Severity.CRITICAL: "CRITICAL",
        Severity.HIGH: "HIGH",
        Severity.MEDIUM: "MEDIUM",
        Severity.LOW: "LOW",
        Severity.INFORMATIONAL: "INFORMATIONAL"
    }
```

```
md = f"""
{severity_emoji.get(finding.severity, "")} {finding.title}
```

```
Severity: {finding.severity.value.upper()}
Status: {finding.status.value}
Location: {finding.location}
```

DESCRIPTION

```
{finding.description}
```

IMPACT

```
{finding.impact}
```

RECOMMENDATION

```
{finding.recommendation}
"""
```

```
if finding.poc:
    md += f"""
```

PROOF OF CONCEPT

```
{finding.poc}
"""
```

```
return md
```

Example Finding

```
critical_finding = FindingWriter(tracker).create_finding(
    title="Reentrancy in withdraw() allows complete fund drainage",
    severity="critical",
    contract="Vault.sol",
    function="withdraw",
    line=142,
    description="""
The withdraw() function sends ETH to the caller before updating
```

their balance. An attacker can deploy a contract that calls `withdraw()` recursively in its `receive()` function, draining all funds from the vault.

The vulnerable code pattern:

```
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient");

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    balances[msg.sender] -= amount; // Updated AFTER external call
}
```

The external call on line 145 transfers control to the caller before the balance is decremented on line 148.

```
""",
    impact = ""
```

Complete loss of all funds in the vault. An attacker can drain the entire contract balance in a single transaction using a flash loan to maximize extraction.

At current vault TVL of \$10M, this represents potential total loss.

```
""",
    recommendation = ""
```

1. Follow Checks-Effects-Interactions pattern - update balance before making external calls:

```
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient");
    balances[msg.sender] -= amount; // Update BEFORE external call

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

2. Add `ReentrancyGuard` modifier for defense in depth:

```
function withdraw(uint256 amount) external nonReentrant {
```

```

// ...
}
"""
    poc_code = """
contract Attacker {
    Vault public vault;
    uint256 public attackCount;

    constructor(address _vault) {
        vault = Vault(_vault);
    }

    function attack() external payable {
        vault.deposit{value: msg.value}();
        vault.withdraw(msg.value);
    }

    receive() external payable {
        if (address(vault).balance >= 1 ether && attackCount < 10) {
            attackCount++;
            vault.withdraw(1 ether);
        }
    }
}
"""
)

```

SECTION 8: FINAL REPORT

Report Structure

class AuditReportGenerator:

```

def __init__(self, project: AuditProject, tracker: AuditTracker):
    self.project = project
    self.tracker = tracker

def generate_executive_summary(self) -> str:
    summary = self.project.summary()

```



```
return f"""
```

EXECUTIVE SUMMARY

Project: {self.project.name}

Repository: {self.project.repository}

Commit: {self.project.commit_hash}

Audit Period: [Start Date] - [End Date]

FINDINGS SUMMARY

Critical: {summary['critical']}

High: {summary['high']}

Medium: {summary['medium']}

Low: {summary['low']}

Informational: {summary['informational']}

OVERALL ASSESSMENT

[Assessment based on findings]

SCOPE

The following contracts were reviewed:

```
{chr(10).join('- ' + s for s in self.project.scope)}
```

OUT OF SCOPE

```
{chr(10).join('- ' + s for s in self.project.out_of_scope)}
```

```
"""
```

```
def generate_methodology_section(self) -> str:
```

```
    return """
```

METHODOLOGY

This audit was conducted using the following methodology:

1. Initial Review

- Documentation review
- Architecture understanding

- Threat modeling

2. Static Analysis

- Slither automated analysis
- Mythril symbolic execution
- Manual pattern matching

3. Manual Review

- Line-by-line code review
- Function-level analysis
- Cross-function interaction analysis
- Business logic verification

4. Dynamic Testing

- Unit test review
- Proof of concept development
- Fuzz testing

5. Report Generation

- Finding documentation
- Severity assessment
- Remediation recommendations

"""

```
def generate_full_report(self) -> str:
    sections = [
        self.generate_executive_summary(),
        self.generate_methodology_section(),
        self.generate_findings_section(),
        self.generate_recommendations_section()
    ]

    return "\n\n".join(sections)

def generate_findings_section(self) -> str:
    finding_writer = FindingWriter(self.tracker)

    sections = []
    sections.append("DETAILED FINDINGS")
```

```

for severity in [Severity.CRITICAL, Severity.HIGH,
Severity.MEDIUM, Severity.LOW, Severity.INFORMATIONAL]:
    findings = self.project.findings_by_severity(severity)

    if findings:
        sections.append(f"\n{severity.value.upper()} SEVERITY FINDINGS\n")

    for i, finding in enumerate(findings, 1):
        sections.append(f"{severity.value[0].upper()}-{i:02d}")
        sections.append(finding_writer.generate_finding_markdown(finding))

    return "\n".join(sections)

def generate_recommendations_section(self) -> str:
    return """
GENERAL RECOMMENDATIONS

1. Implement comprehensive test coverage for all edge cases
2. Add reentrancy guards to all state-modifying external functions
3. Use time-weighted average prices for oracle data
4. Implement slippage protection on all swaps
5. Add emergency pause functionality
6. Consider formal verification for critical math
7. Implement monitoring and alerting for unusual activity
8. Conduct follow-up audit after fixes are implemented
"""

```

This chapter covered the complete audit process from preparation through final report. In the next chapter, we will examine static analysis tools in depth and how to maximize their effectiveness.

BOOK 7: SECURITY AND AUDITING

Chapter 3: Static Analysis Tools

Automated tools find vulnerabilities faster than humans. They scan thousands of lines in seconds. They never get tired. They never miss patterns they know. But they also produce false positives. They miss context. They cannot understand business logic. This chapter teaches you to use static analysis tools effectively while

understanding their limitations.

SECTION 1: UNDERSTANDING STATIC ANALYSIS

What Static Analysis Does

Static analysis examines code without executing it. The tool reads source code or bytecode, builds models of program behavior, and checks for patterns that indicate vulnerabilities.

Types of Analysis:

Pattern Matching:

Looks for known dangerous code patterns. Fast but shallow. Finds common issues but misses novel vulnerabilities.

Data Flow Analysis:

Tracks how data moves through the program. Where does user input go? Can tainted data reach sensitive operations?

Control Flow Analysis:

Maps all possible execution paths. Which functions call which? What conditions lead where?

Symbolic Execution:

Treats variables as symbols instead of concrete values. Explores all possible inputs mathematically. Deep but computationally expensive.

Abstract Interpretation:

Approximates program behavior using abstract domains. Balances precision with performance.

Strengths and Limitations

Strengths:

- Fast: Analyzes entire codebases in minutes
- Consistent: Never forgets to check something
- Scalable: Handles large projects efficiently

- Baseline: Catches common issues before manual review

Limitations:

- False Positives: Reports issues that are not real
- False Negatives: Misses issues tools do not know
- No Context: Cannot understand business requirements
- Pattern Dependent: Only finds what they are programmed to find

SECTION 2: SLITHER DEEP DIVE

Installation and Setup

Slither is the most widely used static analyzer for Solidity:

```
pip install slither-analyzer
```

For development version with latest detectors:

```
pip install slither-analyzer[dev]
```

Verify installation:

```
slither --version
```

Basic Usage

Run Slither on a project:

```
slither .
```

Run on specific contract:

```
slither contracts/Vault.sol
```

Output formats:

```
slither . --json output.json
```

```
slither . --sarif output.sarif
```

```
slither . --markdown-root https://github.com/project/blob/main/
```

Understanding Slither Output

Slither categorizes findings by:

Impact: How bad is this if exploited?

- High: Direct fund loss or critical functionality broken
- Medium: Indirect fund loss or functionality degraded
- Low: Minor issues or best practice violations
- Informational: Code quality and style issues

Confidence: How sure is Slither this is real?

- High: Almost certainly a real issue
- Medium: Likely real but may be false positive
- Low: Possibly real, needs manual verification

Focus on High impact + High confidence first.

Slither Detectors

Slither has over 80 detectors. Key ones for security:

```
from dataclasses import dataclass
from typing import List, Dict, Optional
import subprocess
import json
```

```
@dataclass
class DetectorInfo:
    name: str
    impact: str
    confidence: str
    description: str
    wiki_url: str
```

```
class SlitherDetectorManager:
```

```
    HIGH_PRIORITY_DETECTORS = [
```

```
"reentrancy-eth",  
"reentrancy-no-eth",  
"reentrancy-unlimited-gas",  
"arbitrary-send-eth",  
"arbitrary-send-erc20",  
"controlled-delegatecall",  
"delegatecall-loop",  
"msg-value-loop",  
"protected-vars",  
"unchecked-transfer",  
"unchecked-lowlevel",  
"unchecked-send",  
"uninitialized-storage",  
"uninitialized-local",  
"unused-return",  
"tx-origin",  
"suicidal",  
"shadowing-state",  
"weak-prng",  
"incorrect-equality",  
"locked-ether"  
]
```

```
MEDIUM_PRIORITY_DETECTORS = [  
"divide-before-multiply",  
"reentrancy-benign",  
"reentrancy-events",  
"timestamp",  
"assembly",  
"boolean-cst",  
"constant-function-asm",  
"constant-function-state",  
"deprecated-standards",  
"erc20-interface",  
"erc721-interface",  
"incorrect-modifier",  
"mapping-deletion",  
"missing-zero-check",  
"revert-require",  
"shadowing-abstract",  
]
```

```
"tautology",  
"void-cst"  
]
```

```
def _init_(self, project_path: str):  
    self.project_path = project_path
```

```
def run_high_priority(self) -> Dict:  
    detectors = ",".join(self.HIGH_PRIORITY_DETECTORS)  
    return self._run_with_detectors(detectors)
```

```
def run_medium_priority(self) -> Dict:  
    detectors = ",".join(self.MEDIUM_PRIORITY_DETECTORS)  
    return self._run_with_detectors(detectors)
```

```
def run_all(self) -> Dict:  
    return self._run_with_detectors(None)
```

```
def _run_with_detectors(self, detectors: Optional[str]) -> Dict:  
    cmd = ["slither", self.project_path, "--json", "-"]
```

```
    if detectors:  
        cmd.extend(["--detect", detectors])
```

```
    result = subprocess.run(cmd, capture_output=True, text=True)
```

```
    if result.stdout:  
        return json.loads(result.stdout)  
    return {"results": {"detectors": []}, "error": result.stderr}
```

```
def list_all_detectors(self) -> List[DetectorInfo]:  
    result = subprocess.run(  
        ["slither", "--list-detectors-json"],  
        capture_output=True,  
        text=True  
    )
```

```
    if not result.stdout:  
        return []
```



```

data = json.loads(result.stdout)

detectors = []
for det in data:
    detectors.append(DetectorInfo(
        name=det.get("check", ""),
        impact=det.get("impact", ""),
        confidence=det.get("confidence", ""),
        description=det.get("title", ""),
        wiki_url=det.get("wiki", "")
    ))

return detectors

```

Slither Printers

Printers extract information about code structure:

class SlitherPrinters:

```

def __init__(self, project_path: str):
    self.project_path = project_path

def get_contract_summary(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "contract-summary"],
        capture_output=True,
        text=True
    )
    return result.stdout

def get_function_summary(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "function-summary"],
        capture_output=True,
        text=True
    )
    return result.stdout

```

```
def get_inheritance_graph(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "inheritance-graph"],
        capture_output=True,
        text=True
    )
    return result.stdout
```

```
def get_call_graph(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "call-graph"],
        capture_output=True,
        text=True
    )
    return result.stdout
```

```
def get_human_summary(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "human-summary"],
        capture_output=True,
        text=True
    )
    return result.stdout
```

```
def get_variable_order(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "variable-order"],
        capture_output=True,
        text=True
    )
    return result.stdout
```

```
def get_data_dependency(self) -> str:
    result = subprocess.run(
        ["slither", self.project_path, "--print", "data-dependency"],
        capture_output=True,
        text=True
    )
    return result.stdout
```

Custom Slither Detectors

You can write custom detectors for project-specific patterns:

```
from slither.detectors.abstract_detector import AbstractDetector,
DetectorClassification
from slither.core.declarations import Function
from slither.core.cfg.node import NodeType

class CustomReentrancyDetector(AbstractDetector):
    ARGUMENT = "custom-reentrancy"
    HELP = "Custom reentrancy pattern detector"
    IMPACT = DetectorClassification.HIGH
    CONFIDENCE = DetectorClassification.MEDIUM

    WIKI = "Custom detector for project-specific reentrancy patterns"
    WIKI_TITLE = "Custom Reentrancy"
    WIKI_DESCRIPTION = "Detects reentrancy patterns specific to this
project"
    WIKI_EXPLOIT_SCENARIO = "An attacker could exploit unsafe
external calls"
    WIKI_RECOMMENDATION = "Use ReentrancyGuard or CEI
pattern"

    def _detect(self):
        results = []

        for contract in self.compilation_unit.contracts_derived:
            for function in contract.functions:
                if self._is_vulnerable(function):
                    info = [
                        function,
                        " contains potential reentrancy vulnerability\n"
                    ]
                    results.append(self.generate_result(info))

        return results
```

```

def _is_vulnerable(self, function: Function) -> bool:
    has_external_call = False
    has_state_write_after_call = False
    external_call_found = False

    for node in function.nodes:
        if node.type == NodeType.EXPRESSION:
            if self._contains_external_call(node):
                has_external_call = True
                external_call_found = True

    if external_call_found and self._modifies_state(node):
        has_state_write_after_call = True

    return has_external_call and has_state_write_after_call

def _contains_external_call(self, node) -> bool:
    if node.external_calls_as_expressions:
        return True
    if node.low_level_calls:
        return True
    return False

def _modifies_state(self, node) -> bool:
    if node.state_variables_written:
        return True
    return False

```

Slither Configuration

Configure Slither with `slither.config.json`:

```

{
    "detectors_to_exclude": [
        "naming-convention",
        "solc-version"
    ],
    "exclude_informational": false,
    "exclude_low": false,

```

```
"exclude_medium": false,  
"exclude_high": false,  
"fail_on_info": false,  
"fail_on_low": false,  
"fail_on_medium": true,  
"fail_on_high": true,  
"filter_paths": [  
  "node_modules",  
  "lib",  
  "test",  
  "mock"  
],  
"legacy_ast": false  
}
```

SECTION 3: MYTHRIL DEEP DIVE

Understanding Mythril

Mythril uses symbolic execution. Instead of running code with concrete values, it treats inputs as symbols and explores all possible execution paths mathematically.

Installation:

```
pip install mythril
```

Or use Docker:

```
docker pull mythril/myth  
docker run mythril/myth analyze /contracts/Vault.sol
```

Basic Usage

Analyze a contract:

```
myth analyze contracts/Vault.sol
```

Analyze deployed contract:

```
myth analyze --address 0x123...abc --rpc infura
```

Set execution parameters:

```
myth analyze contracts/Vault.sol --execution-timeout 300 --max-depth 50
```

Mythril analysis takes time. Complex contracts may need hours.

Mythril Modules

Mythril has specialized modules:

```
class MythrilModuleRunner:
```

```
    MODULES = {
        "ether_thief": "Detect unauthorized ETH transfers",
        "suicide": "Detect unprotected selfdestruct",
        "state_change_external_calls": "Detect state changes after external calls",
        "delegatecall": "Detect dangerous delegatecall usage",
        "integer": "Detect integer overflow/underflow",
        "multiple_sends": "Detect multiple ETH sends in one transaction",
        "unchecked_retval": "Detect unchecked return values",
        "user_assertion": "Check user-defined assertions",
        "arbitrary_jump": "Detect arbitrary jump destinations",
        "arbitrary_write": "Detect arbitrary storage writes"
    }
```

```
    def __init__(self, contract_path: str):
        self.contract_path = contract_path
```

```
    def run_module(self, module: str, timeout: int = 300) -> Dict:
        result = subprocess.run(
            [
                "myth", "analyze",
                self.contract_path,
                "--modules", module,
                "--execution-timeout", str(timeout),
```

```

"-o", "json"
],
capture_output=True,
text=True
)

if result.stdout:
    return json.loads(result.stdout)
return {"issues": [], "error": result.stderr}

```

```

def run_security_modules(self) -> Dict:
    security_modules = [
        "ether_thief",
        "suicide",
        "state_change_external_calls",
        "delegatecall",
        "integer"
    ]

    return self.run_module(",".join(security_modules))

```

```

def run_all_modules(self) -> Dict:
    return self.run_module(",".join(self.MODULES.keys()))

```

Interpreting Mythril Output

Mythril provides:

1. Issue description
2. SWCID reference
3. Severity
4. Code location
5. Transaction sequence to trigger the bug

class MythrilResultParser:

```

def __init__(self):
    self.swc_database = {
        "SWC-101": "Integer Overflow and Underflow",

```

```

"SWC-104": "Unchecked Call Return Value",
"SWC-105": "Unprotected Ether Withdrawal",
"SWC-106": "Unprotected SELFDESTRUCT",
"SWC-107": "Reentrancy",
"SWC-110": "Assert Violation",
"SWC-112": "Delegatecall to Untrusted Callee",
"SWC-113": "DoS with Failed Call",
"SWC-114": "Transaction Order Dependence",
"SWC-115": "Authorization through tx.origin",
"SWC-116": "Block values as a proxy for time",
"SWC-120": "Weak Sources of Randomness",
"SWC-124": "Write to Arbitrary Storage Location",
"SWC-127": "Arbitrary Jump with Function Type Variable",
"SWC-128": "DoS With Block Gas Limit",
"SWC-131": "Presence of unused variables",
"SWC-132": "Unexpected Ether balance",
"SWC-134": "Message call with hardcoded gas amount",
"SWC-136": "Unencrypted Private Data On-Chain"
}

```

```

def parse_results(self, mythrill_output: Dict) -> List[Dict]:
    parsed = []

```

```

    for issue in mythrill_output.get("issues", []):
        parsed_issue = {
            "title": issue.get("title", "Unknown"),
            "swc_id": issue.get("swc-id", ""),
            "swc_name": self.swc_database.get(issue.get("swc-id", ""),
            "Unknown"),
            "severity": issue.get("severity", "Unknown"),
            "description": issue.get("description", ""),
            "contract": issue.get("contract", ""),
            "function": issue.get("function", ""),
            "address": issue.get("address", ""),
            "source_mapping": issue.get("sourceMap", ""),
            "tx_sequence": issue.get("tx_sequence", {})
        }

```

```

    parsed.append(parsed_issue)

```



```
return parsed
```

```
def extract_poc_transactions(self, issue: Dict) -> List[Dict]:  
    tx_sequence = issue.get("tx_sequence", {})  
    transactions = []
```

```
    for step in tx_sequence.get("steps", []):
```

```
        tx = {  
            "caller": step.get("origin", ""),  
            "input": step.get("input", ""),  
            "value": step.get("value", "0"),  
            "call_data": step.get("calldata", "")  
        }
```

```
        transactions.append(tx)
```

```
    return transactions
```

SECTION 4: ECHIDNA FUZZING

Understanding Echidna

Echidna is a property-based fuzzer. You define properties that should always be true. Echidna tries to find inputs that violate them.

Installation:

```
brew install echidna
```

```
pip install echidna
```

Or download binary from GitHub releases.

Writing Echidna Tests

Properties are Solidity functions that return bool:

```
contract EchidnaVaultTest {  
    Vault public vault;  
    uint256 public totalDeposited;
```

```
constructor() {  
    vault = new Vault();  
}
```

```
function deposit(uint256 amount) public payable {  
    vault.deposit{value: amount}();  
    totalDeposited += amount;  
}
```

```
function withdraw(uint256 amount) public {  
    uint256 balanceBefore = address(vault).balance;
```

```
    try vault.withdraw(amount) {  
        totalDeposited -= amount;  
    } catch {  
    }  
}
```

```
function echidna_balance_consistency() public view returns (bool) {  
    return address(vault).balance >= 0;  
}
```

```
function echidna_no_free_money() public view returns (bool) {  
    return address(this).balance <= totalDeposited;  
}
```

```
function echidna_total_matches_balance() public view returns  
(bool) {  
    return vault.totalDeposits() == address(vault).balance;  
}  
}
```

Properties must:

- Start with echidna_ prefix
- Return bool
- Take no arguments
- Be view or pure

Running Echidna:

```
echidna contracts/EchidnaVaultTest.sol --contract EchidnaVaultTest
```

Echidna Configuration

Configure with echidna.yaml:

```
testMode: "property"
testLimit: 50000
timeout: 300
shrinkLimit: 5000
seqLen: 100
contractAddr: "0xdeadbeef"
sender: ["0x1", "0x2", "0x3"]
deployer: "0x1"
initialBalance: 1000000000000000000000
coverage: true
corpusDir: "corpus"
```

Python wrapper for Echidna:

```
from dataclasses import dataclass
from typing import List, Dict, Optional
import subprocess
import yaml
```

```
@dataclass
class EchidnaConfig:
    test_mode: str = "property"
    test_limit: int = 50000
    timeout: int = 300
    shrink_limit: int = 5000
    seq_len: int = 100
    coverage: bool = True
    corpus_dir: str = "corpus"
    senders: List[str] = None
    initial_balance: int = 100 * 10**18
```

```
def to_yaml(self) -> str:
```

```

config = {
    "testMode": self.test_mode,
    "testLimit": self.test_limit,
    "timeout": self.timeout,
    "shrinkLimit": self.shrink_limit,
    "seqLen": self.seq_len,
    "coverage": self.coverage,
    "corpusDir": self.corpus_dir,
    "initialBalance": self.initial_balance
}

```

```

if self.senders:
    config["sender"] = self.senders

```

```

return yaml.dump(config)

```

```

class EchidnaRunner:

```

```

    def __init__(self, contract_path: str, contract_name: str):
        self.contract_path = contract_path
        self.contract_name = contract_name

```

```

    def run(
        self,
        config: Optional[EchidnaConfig] = None,
        config_file: Optional[str] = None
    ) -> Dict:

```

```

        cmd = ["echidna", self.contract_path, "--contract",
            self.contract_name]

```

```

        if config_file:
            cmd.extend(["--config", config_file])
        elif config:
            config_path = "/tmp/echidna_config.yaml"
            with open(config_path, "w") as f:
                f.write(config.to_yaml())
            cmd.extend(["--config", config_path])

```

```
cmd.append("--format")
cmd.append("json")
```

```
result = subprocess.run(cmd, capture_output=True, text=True)
```

```
if result.stdout:
    import json
    return json.loads(result.stdout)
return {"error": result.stderr}
```

```
def run_quick_test(self) -> Dict:
    config = EchidnaConfig(
        test_limit=10000,
        timeout=60,
        seq_len=50
    )
    return self.run(config=config)
```

```
def run_thorough_test(self) -> Dict:
    config = EchidnaConfig(
        test_limit=100000,
        timeout=600,
        seq_len=200,
        coverage=True
    )
    return self.run(config=config)
```

Advanced Echidna Patterns

Test specific attack scenarios:

```
contract EchidnaReentrancyTest {
    Vault public vault;
    bool public reentrancyAttempted;
    bool public reentrancySucceeded;

    constructor() payable {
        vault = new Vault();
    }
}
```

```

function depositAndWithdraw() public payable {
    if (msg.value > 0) {
        vault.deposit{value: msg.value}();
        vault.withdraw(msg.value);
    }
}

```

```

receive() external payable {
    if (!reentrancyAttempted && address(vault).balance > 0) {
        reentrancyAttempted = true;

```

```

        try vault.withdraw(address(vault).balance) {
            reentrancySucceeded = true;
        } catch {
        }
    }
}

```

```

function echidna_no_reentrancy_possible() public view returns
(bool) {
    return !reentrancySucceeded;
}
}

```

SECTION 5: FOUNDRY TESTING

Foundry Fuzz Tests

Foundry has built-in fuzzing:

```

contract VaultFuzzTest is Test {
    Vault vault;

    function setUp() public {
        vault = new Vault();
        vm.deal(address(this), 1000 ether);
    }
}

```

```
function testFuzz_DepositWithdraw(uint256 amount) public {  
    amount = bound(amount, 0.01 ether, 100 ether);
```

```
    vault.deposit{value: amount}();  
    assertEq(vault.balances(address(this)), amount);
```

```
    vault.withdraw(amount);  
    assertEq(vault.balances(address(this)), 0);  
}
```

```
function testFuzz_NoFreeWithdrawals(uint256 depositAmount,  
uint256 withdrawAmount) public {  
    depositAmount = bound(depositAmount, 0.01 ether, 100 ether);
```

```
    vault.deposit{value: depositAmount}();
```

```
    if (withdrawAmount > depositAmount) {  
        vm.expectRevert("Insufficient balance");  
        vault.withdraw(withdrawAmount);  
    } else {  
        vault.withdraw(withdrawAmount);  
        assertEq(vault.balances(address(this)), depositAmount -  
withdrawAmount);  
    }  
}
```

```
function testFuzz_MultipleUsers(address user1, address user2,  
uint256 amount1, uint256 amount2) public {  
    vm.assume(user1 != address(0) && user2 != address(0));  
    vm.assume(user1 != user2);  
    amount1 = bound(amount1, 0.01 ether, 50 ether);  
    amount2 = bound(amount2, 0.01 ether, 50 ether);
```

```
    vm.deal(user1, amount1);  
    vm.deal(user2, amount2);
```

```
    vm.prank(user1);  
    vault.deposit{value: amount1}();
```

```
    vm.prank(user2);
```

```

vault.deposit{value: amount2}();

assertEq(address(vault).balance, amount1 + amount2);

vm.prank(user1);
vault.withdraw(amount1);

assertEq(vault.balances(user1), 0);
assertEq(vault.balances(user2), amount2);
}
}

```

Foundry Invariant Tests

Invariants are properties that must always hold:

```

contract VaultInvariantTest is Test {
    Vault vault;
    VaultHandler handler;

    function setUp() public {
        vault = new Vault();
        handler = new VaultHandler(vault);

        bytes4[] memory selectors = new bytes4[](2);
        selectors[0] = VaultHandler.deposit.selector;
        selectors[1] = VaultHandler.withdraw.selector;

        targetSelector(FuzzSelector({addr: address(handler), selectors:
selectors})));
        targetContract(address(handler));
    }

    function invariant_solventVault() public {
        assertGe(address(vault).balance, 0);
    }

    function invariant_totalMatchesBalance() public {
        assertEq(vault.totalDeposits(), address(vault).balance);
    }
}

```



```

function invariant_noNegativeBalances() public {
    address[] memory users = handler.getUsers();
    for (uint256 i = 0; i < users.length; i++) {
        assertGe(vault.balances(users[i]), 0);
    }
}
}

```

```

contract VaultHandler is Test {
    Vault vault;
    address[] public users;
    mapping(address => bool) public isUser;

    constructor(Vault _vault) {
        vault = _vault;
    }
}

```

```

function deposit(uint256 amount) public {
    amount = bound(amount, 0.01 ether, 10 ether);
}

```

```

    address user = msg.sender;
    if (!isUser[user]) {
        users.push(user);
        isUser[user] = true;
    }
}

```

```

    vm.deal(user, amount);
    vm.prank(user);
    vault.deposit{value: amount}();
}

```

```

function withdraw(uint256 amount) public {
    address user = msg.sender;
    uint256 balance = vault.balances(user);
}

```

```

    if (balance == 0) return;
}

```

```

    amount = bound(amount, 1, balance);
}

```

```

vm.prank(user);
vault.withdraw(amount);
}

```

```

function getUsers() external view returns (address[] memory) {
    return users;
}
}

```

SECTION 6: TOOL INTEGRATION PIPELINE

Comprehensive Analysis Pipeline

```

class ComprehensiveAnalysisPipeline:

```

```

    def __init__(self, project_path: str):
        self.project_path = project_path
        self.results = {}

```

```

    def run_all_tools(self) -> Dict[str, any]:
        self.results["slither"] = self._run_slither()
        self.results["mythril"] = self._run_mythril()
        self.results["pattern_scan"] = self._run_pattern_scan()

```

```

        self.results["summary"] = self._generate_summary()

```

```

    return self.results

```

```

    def _run_slither(self) -> Dict:
        try:
            manager = SlitherDetectorManager(self.project_path)
            findings = manager.run_all()

```

```

            high_findings = []
            medium_findings = []

```

```

            for detector in findings.get("results", {}).get("detectors", []):
                if detector.get("impact") == "High":
                    high_findings.append(detector)

```

```
elif detector.get("impact") == "Medium":  
    medium_findings.append(detector)
```

```
    return {  
        "status": "success",  
        "high": high_findings,  
        "medium": medium_findings,  
        "total": len(findings.get("results", {})).get("detectors", [])  
    }
```

```
except Exception as e:  
    return {"status": "error", "message": str(e)}
```

```
def _run_mythril(self) -> Dict:  
    try:  
        import glob
```

```
        sol_files = glob.glob(f'{self.project_path}/**/*.sol',  
                                recursive=True)  
        all_issues = []
```

```
        for sol_file in sol_files[:5]:  
            runner = MythrilModuleRunner(sol_file)  
            result = runner.run_security_modules()  
            all_issues.extend(result.get("issues", []))
```

```
    return {  
        "status": "success",  
        "issues": all_issues,  
        "total": len(all_issues)  
    }
```

```
except Exception as e:  
    return {"status": "error", "message": str(e)}
```

```
def _run_pattern_scan(self) -> Dict:  
    try:  
        import glob
```

```
        matcher = PatternMatcher()
```

```
all_findings = {}
```

```
sol_files = glob.glob(f'{self.project_path}/**/*.sol',  
recursive=True)
```

```
for sol_file in sol_files:  
    with open(sol_file, "r") as f:  
        code = f.read()
```

```
findings = matcher.scan_code(code)  
if findings:  
    all_findings[sol_file] = findings
```

```
return {  
    "status": "success",  
    "findings": all_findings  
}
```

```
except Exception as e:  
    return {"status": "error", "message": str(e)}
```

```
def _generate_summary(self) -> Dict:  
    summary = {  
        "total_issues": 0,  
        "critical": 0,  
        "high": 0,  
        "medium": 0,  
        "low": 0,  
        "tools_run": []  
    }
```

```
if self.results.get("slither", {}).get("status") == "success":  
    summary["tools_run"].append("slither")  
    summary["high"] += len(self.results["slither"].get("high", []))  
    summary["medium"] += len(self.results["slither"].get("medium",  
[]))
```

```
if self.results.get("mythril", {}).get("status") == "success":  
    summary["tools_run"].append("mythril")  
    for issue in self.results["mythril"].get("issues", []):
```

```

severity = issue.get("severity", "").lower()
if severity in summary:
    summary[severity] += 1

summary["total_issues"] = (
    summary["critical"] +
    summary["high"] +
    summary["medium"] +
    summary["low"]
)

return summary

```

```

class PatternMatcher:

```

```

    def __init__(self):
        self.patterns = {
            "reentrancy": [
                (r"\.call\{.*value:", "Low-level call with value"),
                (r"\.transfer\(", "Transfer before state update"),
            ],
            "access_control": [
                (r"tx\.origin", "tx.origin used"),
            ],
            "oracle": [
                (r"getReserves\(\)", "Spot price from reserves"),
            ]
        }

```

```

    def scan_code(self, code: str) -> Dict[str, List]:
        import re

```

```

        findings = {}
        lines = code.split("\n")

```

```

        for category, patterns in self.patterns.items():
            category_findings = []

```

```

        for pattern, description in patterns:

```

```
for line_num, line in enumerate(lines, 1):
    if re.search(pattern, line):
        category_findings.append({
            "description": description,
            "line": line_num,
            "code": line.strip()
        })

if category_findings:
    findings[category] = category_findings

return findings
```

CI/CD Integration

Integrate security tools into your pipeline:

name: Security Analysis

on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
slither:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3

- name: Install dependencies
run: npm install

- name: Run Slither
uses: crytic/slither-action@v0.3.0
with:
node-version: 18
fail-on: medium

mythril:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Run Mythril

run: |

pip install mythril

myth analyze contracts/*.sol --execution-timeout 300

echidna:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Install Echidna

run: |

pip install solc-select

solc-select install 0.8.19

solc-select use 0.8.19

pip install echidna

- name: Run Echidna

run: echidna contracts/test/EchidnaTest.sol --contract EchidnaTest

--test-limit 10000

SECTION 7: FALSE POSITIVE MANAGEMENT

Understanding False Positives

Tools report issues that are not real vulnerabilities:

1. Context Missing: Tool does not understand the business logic
2. Mitigation Elsewhere: Protection exists but tool does not see it
3. Intended Behavior: Code does what developer wanted
4. Unreachable: Vulnerable code path cannot be executed

False Positive Tracking

```
class FalsePositiveManager:
```

```
    def __init__(self, findings_path: str):  
        self.findings_path = findings_path  
        self.known_false_positives = self._load_false_positives()
```

```
    def _load_false_positives(self) -> List[Dict]:  
        try:  
            with open(f"{self.findings_path}/false_positives.json", "r") as f:  
                return json.load(f)  
        except FileNotFoundError:  
            return []
```

```
    def is_false_positive(self, finding: Dict) -> bool:  
        for fp in self.known_false_positives:  
            if self._matches(finding, fp):  
                return True  
        return False
```

```
    def _matches(self, finding: Dict, fp: Dict) -> bool:  
        if fp.get("detector") != finding.get("check"):  
            return False
```

```
        if fp.get("location") and fp.get("location") !=  
            finding.get("location"):  
            return False
```

```
        if fp.get("description_contains"):  
            if fp["description_contains"] not in finding.get("description", ""):  
                return False
```

```
        return True
```

```
    def mark_false_positive(  
        self,  
        finding: Dict,  
        reason: str,  
        verified_by: str  
    ) -> None:
```



```

fp_entry = {
"detector": finding.get("check"),
"location": finding.get("location"),
"description_contains": finding.get("description", "")[:100],
"reason": reason,
"verified_by": verified_by,
"date": str(datetime.now())
}

self.known_false_positives.append(fp_entry)
self._save_false_positives()

def _save_false_positives(self) -> None:
with open(f'{self.findings_path}/false_positives.json', "w") as f:
json.dump(self.known_false_positives, f, indent=2)

def filter_findings(self, findings: List[Dict]) -> List[Dict]:
return [f for f in findings if not self.is_false_positive(f)]

```

from datetime import datetime

Inline Suppressions

Suppress false positives in code:

```

contract Vault {
function withdraw(uint256 amount) external nonReentrant {
require(balances[msg.sender] >= amount, "Insufficient");

balances[msg.sender] -= amount;

(bool success, ) = msg.sender.call{value: amount}("");
require(success, "Transfer failed");
}
}

```

SECTION 8: TOOL LIMITATIONS

What Tools Miss

1. Business Logic Bugs

Tools do not understand what code should do:

```
function distributeRewards(address[] memory users, uint256[]  
memory amounts) external {  
    for (uint256 i = 0; i < users.length; i++) {  
        rewards[users[i]] += amounts[i];  
    }  
}
```

Tools see nothing wrong. But if amounts should sum to totalRewards, there is no check.

2. Complex Multi-Transaction Attacks

Tools analyze single transactions or short sequences. Attacks spanning many transactions or requiring specific timing may be missed.

3. Economic Exploits

Flash loan attacks, oracle manipulation, governance attacks require understanding of DeFi economics.

4. Cross-Contract Vulnerabilities

Interactions between multiple contracts may create vulnerabilities neither has alone.

Compensating for Limitations

```
class ManualCheckGenerator:
```

```
    def __init__(self, contract_analysis: Dict):  
        self.analysis = contract_analysis
```

```
    def generate_manual_checklist(self) -> List[str]:  
        checks = []
```

```
        if self.analysis.get("has_token_transfers"):
```

```
checks.append("Verify token transfer logic matches business
requirements")
checks.append("Check for fee-on-transfer token compatibility")
checks.append("Verify rebasing token handling")

if self.analysis.get("has_external_calls"):
checks.append("Map all external call trust assumptions")
checks.append("Check for callback attack vectors")
checks.append("Verify external contract addresses are validated")

if self.analysis.get("has_price_calculations"):
checks.append("Verify price oracle manipulation resistance")
checks.append("Check flash loan attack vectors")
checks.append("Validate price deviation bounds")

if self.analysis.get("has_time_dependencies"):
checks.append("Check timestamp manipulation windows")
checks.append("Verify time-dependent logic edge cases")

if self.analysis.get("has_access_control"):
checks.append("Map all privileged operations")
checks.append("Verify role separation")
checks.append("Check for centralization risks")

if self.analysis.get("has_upgradability"):
checks.append("Verify storage layout compatibility")
checks.append("Check for function selector clashes")
checks.append("Validate upgrade authorization")

return checks
```

This chapter covered major static analysis tools and how to use them effectively. Tools are essential but not sufficient. The next chapter covers runtime analysis and dynamic testing techniques that complement static analysis.

BOOK 7: SECURITY AND AUDITING

Chapter 4: Runtime Analysis and Testing

Static analysis examines code without running it. Runtime analysis

executes code and observes behavior. You see actual state changes. You trace real transactions. You catch bugs that only manifest during execution. This chapter teaches you to build comprehensive testing frameworks that catch vulnerabilities before deployment.

SECTION 1: TESTING PHILOSOPHY

Why Testing Matters

Smart contracts are immutable. Once deployed, bugs cannot be patched. Testing is your last line of defense before permanent deployment. Every hour spent testing saves potential millions in losses.

Testing is not about proving code works. Testing is about trying to break code. You write tests that attack your own contracts. You think like an adversary. You assume everything can fail.

Testing Pyramid for Smart Contracts

Unit Tests:

Test individual functions in isolation. Fast. Cover edge cases. Should have highest coverage.

Integration Tests:

Test contract interactions. Multiple contracts working together. State transitions across calls.

Fuzz Tests:

Random inputs to find edge cases. Property-based testing. Let the computer find bugs.

Invariant Tests:

Properties that must always hold. System-wide constraints. Run across many transactions.

Fork Tests:

Test against real mainnet state. Verify behavior with actual deployed contracts. Catch integration issues.

Formal Verification:

Mathematical proofs of correctness. Highest assurance. Most expensive.

SECTION 2: FOUNDRY TESTING FRAMEWORK

Setting Up Foundry

Foundry is the modern standard for Solidity testing:

```
curl -L https://foundry.paradigm.xyz | bash
foundryup
```

Create new project:

```
forge init my-project
cd my-project
```

Project structure:

```
my-project/
src/
Contract.sol
test/
Contract.t.sol
script/
Deploy.s.sol
foundry.toml
```

Basic Test Structure

```
import {Test, console} from "forge-std/Test.sol";
import {Vault} from "../src/Vault.sol";
```

```
contract VaultTest is Test {
    Vault public vault;
    address public owner;
    address public user1;
```

```
address public user2;
```

```
function setUp() public {  
  owner = makeAddr("owner");  
  user1 = makeAddr("user1");  
  user2 = makeAddr("user2");
```

```
  vm.deal(owner, 100 ether);  
  vm.deal(user1, 100 ether);  
  vm.deal(user2, 100 ether);
```

```
  vm.prank(owner);  
  vault = new Vault();  
}
```

```
function test_Deposit() public {  
  vm.prank(user1);  
  vault.deposit{value: 1 ether}();
```

```
  assertEq(vault.balances(user1), 1 ether);  
  assertEq(address(vault).balance, 1 ether);  
}
```

```
function test_Withdraw() public {  
  vm.startPrank(user1);  
  vault.deposit{value: 5 ether}();
```

```
  uint256 balanceBefore = user1.balance;  
  vault.withdraw(3 ether);  
  uint256 balanceAfter = user1.balance;
```

```
  assertEq(balanceAfter - balanceBefore, 3 ether);  
  assertEq(vault.balances(user1), 2 ether);  
  vm.stopPrank();  
}
```

```
function test_RevertWhen_WithdrawMoreThanBalance() public {  
  vm.startPrank(user1);  
  vault.deposit{value: 1 ether}();
```

```
vm.expectRevert("Insufficient balance");
vault.withdraw(2 ether);
vm.stopPrank();
}
```

```
function test_RevertWhen_WithdrawWithZeroBalance() public {
    vm.prank(user1);
    vm.expectRevert("Insufficient balance");
    vault.withdraw(1 ether);
}
}
```

Foundry Cheatcodes

Cheatcodes manipulate the EVM for testing:

```
contract CheatcodeExamples is Test {
    function test_PrankExample() public {
        vm.prank(address(0x123));
    }
}
```

```
function test_StartPrankExample() public {
    vm.startPrank(address(0x123));
    vm.stopPrank();
}
```

```
function test_DealExample() public {
    address target = address(0x456);
    vm.deal(target, 100 ether);
    assertEq(target.balance, 100 ether);
}
```

```
function test_WarpExample() public {
    vm.warp(block.timestamp + 1 days);
}
```

```
function test_RollExample() public {
    vm.roll(block.number + 100);
}
```

```
function test_ExpectRevertExample() public {  
    vm.expectRevert("Error message");  
}
```

```
function test_ExpectEmitExample() public {  
    vm.expectEmit(true, true, false, true);  
}
```

```
function test_SnapshotExample() public {  
    uint256 snapshotId = vm.snapshot();
```

```
    vm.revertTo(snapshotId);  
}
```

```
function test_LabelExample() public {  
    address target = address(0x789);  
    vm.label(target, "ImportantContract");  
}
```

```
function test_MakeAddrExample() public {  
    address newUser = makeAddr("uniqueUser");  
    assertNotEq(newUser, address(0));  
}
```

```
function test_SignExample() public {  
    uint256 privateKey = 0xabc123;  
    address signer = vm.addr(privateKey);
```

```
    bytes32 digest = keccak256("message");  
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(privateKey, digest);
```

```
    address recovered = ecrecover(digest, v, r, s);  
    assertEq(recovered, signer);  
}  
}
```

SECTION 3: SECURITY-FOCUSED TEST PATTERNS

Reentrancy Attack Tests


```

contract ReentrancyTest is Test {
    Vault public vault;
    ReentrancyAttacker public attacker;

    function setUp() public {
        vault = new Vault();

        vm.deal(address(this), 100 ether);
        vault.deposit{value: 50 ether}();

        attacker = new ReentrancyAttacker(address(vault));
        vm.deal(address(attacker), 10 ether);
    }

    function test_ReentrancyAttackFails() public {
        uint256 vaultBalanceBefore = address(vault).balance;

        vm.expectRevert();
        attacker.attack{value: 1 ether}();

        assertEq(address(vault).balance, vaultBalanceBefore);
    }

    function test_ReentrancyGuardPreventsAttack() public {
        attacker.attack{value: 1 ether}();

        assertLe(attacker.attackCount(), 1);
    }
}

contract ReentrancyAttacker {
    Vault public vault;
    uint256 public attackCount;

    constructor(address _vault) {
        vault = Vault(_vault);
    }

    function attack() external payable {

```

```

vault.deposit{value: msg.value}();
vault.withdraw(msg.value);
}

```

```

receive() external payable {
if (address(vault).balance >= 1 ether && attackCount < 10) {
attackCount ++;
vault.withdraw(1 ether);
}
}
}

```

Access Control Tests

```

contract AccessControlTest is Test {
    ProtectedVault public vault;
    address public owner;
    address public admin;
    address public user;
    address public attacker;

    function setUp() public {
        owner = makeAddr("owner");
        admin = makeAddr("admin");
        user = makeAddr("user");
        attacker = makeAddr("attacker");

        vm.prank(owner);
        vault = new ProtectedVault();

        vm.prank(owner);
        vault.grantRole(vault.ADMIN_ROLE(), admin);
    }

    function test_OnlyOwnerCanSetFee() public {
        vm.prank(owner);
        vault.setFee(100);
        assertEq(vault.fee(), 100);
    }
}

```

```
function test_AdminCannotSetFee() public {
    vm.prank(admin);
    vm.expectRevert();
    vault.setFee(100);
}
```

```
function test_UserCannotSetFee() public {
    vm.prank(user);
    vm.expectRevert();
    vault.setFee(100);
}
```

```
function test_AttackerCannotBypassAccessControl() public {
    vm.prank(attacker);
    vm.expectRevert();
    vault.emergencyWithdraw();
```

```
    vm.prank(attacker);
    vm.expectRevert();
    vault.pause();
```

```
    vm.prank(attacker);
    vm.expectRevert();
    vault.grantRole(vault.ADMIN_ROLE(), attacker);
}
```

```
function test_RoleRenunciationWorks() public {
    vm.prank(admin);
    vault.renounceRole(vault.ADMIN_ROLE(), admin);
```

```
    assertFalse(vault.hasRole(vault.ADMIN_ROLE(), admin));
}
}
```

Integer Overflow Tests

```
contract OverflowTest is Test {
    MathContract public math;
```

```
    function setUp() public {
```

```
math = new MathContract();  
}
```

```
function test_AdditionOverflow() public {  
    uint256 a = type(uint256).max;  
    uint256 b = 1;
```

```
    vm.expectRevert();  
    math.add(a, b);  
}
```

```
function test_MultiplicationOverflow() public {  
    uint256 a = type(uint256).max / 2 + 1;  
    uint256 b = 2;
```

```
    vm.expectRevert();  
    math.multiply(a, b);  
}
```

```
function test_SubtractionUnderflow() public {  
    uint256 a = 0;  
    uint256 b = 1;
```

```
    vm.expectRevert();  
    math.subtract(a, b);  
}
```

```
function test_DivisionByZero() public {  
    uint256 a = 100;  
    uint256 b = 0;
```

```
    vm.expectRevert();  
    math.divide(a, b);  
}
```

```
function test_SafeMathEdgeCases() public {  
    assertEq(math.add(0, 0), 0);  
    assertEq(math.multiply(0, type(uint256).max), 0);  
    assertEq(math.subtract(type(uint256).max, type(uint256).max), 0);  
}
```

```
}
```

Oracle Manipulation Tests

```
contract OracleManipulationTest is Test {
    LendingProtocol public lending;
    MockOracle public oracle;
    IERC20 public collateralToken;
    IERC20 public loanToken;

    function setUp() public {
        oracle = new MockOracle();
        collateralToken = new MockERC20("Collateral", "COL");
        loanToken = new MockERC20("Loan", "LOAN");

        lending = new LendingProtocol(
            address(oracle),
            address(collateralToken),
            address(loanToken)
        );

        MockERC20(address(loanToken)).mint(address(lending), 1000000
ether);
    }

    function test_OracleManipulationAttackFails() public {
        address attacker = makeAddr("attacker");
        MockERC20(address(collateralToken)).mint(attacker, 100 ether);

        vm.startPrank(attacker);
        collateralToken.approve(address(lending), 100 ether);
        lending.deposit(100 ether);

        oracle.setPrice(1 ether);
        uint256 normalBorrow = lending.maxBorrowable(attacker);

        oracle.setPrice(100 ether);

        vm.expectRevert("Price deviation too high");
        lending.borrow(normalBorrow * 50);
    }
}
```

```
vm.stopPrank();  
}
```

```
function test_TWAPResistsManipulation() public {  
    oracle.setPrice(1 ether);
```

```
    vm.warp(block.timestamp + 1 hours);  
    oracle.setPrice(100 ether);
```

```
    uint256 twapPrice = oracle.getTWAP();
```

```
    assertLt(twapPrice, 50 ether);  
}  
}
```

```
contract MockOracle {  
    uint256 public price = 1 ether;  
    uint256[] public priceHistory;  
    uint256[] public timestamps;
```

```
    function setPrice(uint256 _price) external {  
        priceHistory.push(price);  
        timestamps.push(block.timestamp);  
        price = _price;  
    }
```

```
    function latestAnswer() external view returns (uint256) {  
        return price;  
    }
```

```
    function getTWAP() external view returns (uint256) {  
        if (priceHistory.length == 0) return price;
```

```
        uint256 sum = 0;  
        for (uint256 i = 0; i < priceHistory.length; i++) {  
            sum += priceHistory[i];  
        }  
        return sum / priceHistory.length;  
    }
```

```
}
```

SECTION 4: FUZZ TESTING

Basic Fuzz Tests

```
contract FuzzTest is Test {  
    Token public token;
```

```
    function setUp() public {  
        token = new Token("Test", "TST", 1000000 ether);  
    }
```

```
    function testFuzz_Transfer(address to, uint256 amount) public {  
        vm.assume(to != address(0));  
        vm.assume(to != address(this));  
        amount = bound(amount, 0, token.balanceOf(address(this)));
```

```
        uint256 senderBefore = token.balanceOf(address(this));  
        uint256 receiverBefore = token.balanceOf(to);
```

```
        token.transfer(to, amount);
```

```
        assertEq(token.balanceOf(address(this)), senderBefore - amount);  
        assertEq(token.balanceOf(to), receiverBefore + amount);  
    }
```

```
    function testFuzz_Approve(address spender, uint256 amount)  
    public {  
        vm.assume(spender != address(0));
```

```
        token.approve(spender, amount);
```

```
        assertEq(token.allowance(address(this), spender), amount);  
    }
```

```
    function testFuzz_TransferFrom(  
        address from,  
        address to,
```

```

uint256 approveAmount,
uint256 transferAmount
) public {
vm.assume(from != address(0));
vm.assume(to != address(0));
vm.assume(from != to);

deal(address(token), from, approveAmount);

transferAmount = bound(transferAmount, 0, approveAmount);

vm.prank(from);
token.approve(address(this), approveAmount);

uint256 fromBefore = token.balanceOf(from);
uint256 toBefore = token.balanceOf(to);

token.transferFrom(from, to, transferAmount);

assertEq(token.balanceOf(from), fromBefore - transferAmount);
assertEq(token.balanceOf(to), toBefore + transferAmount);
}
}

```

Advanced Fuzz Patterns

```

contract AdvancedFuzzTest is Test {
    Vault public vault;

    function setUp() public {
        vault = new Vault();
    }

    function testFuzz_DepositWithdrawIntegrity(
        uint256[] memory deposits,
        uint256[] memory withdrawals
    ) public {
        vm.assume(deposits.length > 0);
        vm.assume(deposits.length <= 10);
        vm.assume(withdrawals.length <= deposits.length);
    }
}

```



```
uint256 totalDeposited = 0;
```

```
for (uint256 i = 0; i < deposits.length; i++) {  
    deposits[i] = bound(deposits[i], 0.01 ether, 10 ether);  
    vm.deal(address(this), deposits[i]);  
    vault.deposit{value: deposits[i]}();  
    totalDeposited += deposits[i];  
}
```

```
assertEq(vault.balances(address(this)), totalDeposited);
```

```
uint256 totalWithdrawn = 0;  
for (uint256 i = 0; i < withdrawals.length; i++) {  
    uint256 maxWithdraw = vault.balances(address(this));  
    if (maxWithdraw == 0) break;
```

```
    withdrawals[i] = bound(withdrawals[i], 1, maxWithdraw);  
    vault.withdraw(withdrawals[i]);  
    totalWithdrawn += withdrawals[i];  
}
```

```
assertEq(vault.balances(address(this)), totalDeposited -  
totalWithdrawn);  
}
```

```
function testFuzz_MultiUserInteractions(  
    uint8 numUsers,  
    uint256 seed  
) public {  
    numUsers = uint8(bound(numUsers, 2, 10));
```

```
    address[] memory users = new address[](numUsers);  
    uint256[] memory expectedBalances = new uint256[](numUsers);
```

```
    for (uint8 i = 0; i < numUsers; i++) {  
        users[i] = address(uint160(uint256(keccak256(abi.encode(seed,  
i))))));  
        uint256 depositAmount = bound(  
            uint256(keccak256(abi.encode(seed, i, "deposit"))),
```

```

0.1 ether,
10 ether
);

vm.deal(users[i], depositAmount);
vm.prank(users[i]);
vault.deposit{value: depositAmount}();

expectedBalances[i] = depositAmount;
}

for (uint8 i = 0; i < numUsers; i++) {
    assertEq(vault.balances(users[i]), expectedBalances[i]);
}

uint256 totalExpected = 0;
for (uint8 i = 0; i < numUsers; i++) {
    totalExpected += expectedBalances[i];
}
assertEq(address(vault).balance, totalExpected);
}
}

```

SECTION 5: INVARIANT TESTING

Defining Invariants

Invariants are properties that must always be true regardless of what operations are performed:

```

contract VaultInvariantTest is Test {
    Vault public vault;
    VaultHandler public handler;

    function setUp() public {
        vault = new Vault();
        handler = new VaultHandler(vault);

        targetContract(address(handler));
    }
}

```

```
}
```

```
function invariant_SolvencyCheck() public view {  
    assertGe(  
        address(vault).balance,  
        vault.totalDeposits(),  
        "Vault is insolvent"  
    );  
}
```

```
function invariant_BalancesMatchTotal() public view {  
    uint256 sumOfBalances = handler.sumOfAllBalances();  
    assertEq(  
        sumOfBalances,  
        vault.totalDeposits(),  
        "Individual balances do not match total"  
    );  
}
```

```
function invariant_NoNegativeBalances() public view {  
    address[] memory users = handler.getActors();  
    for (uint256 i = 0; i < users.length; i++) {  
        assertGe(vault.balances(users[i]), 0, "Negative balance detected");  
    }  
}
```

```
function invariant_CallSummary() public view {  
    handler.callSummary();  
}  
}
```

```
contract VaultHandler is Test {  
    Vault public vault;
```

```
    address[] public actors;  
    mapping(address => bool) public isActor;
```

```
    uint256 public depositCalls;  
    uint256 public withdrawCalls;  
    uint256 public depositSuccess;
```

```
uint256 public withdrawSuccess;
```

```
modifier createActor() {  
    address actor = msg.sender;  
    if (!isActor[actor]) {  
        actors.push(actor);  
        isActor[actor] = true;  
    }  
    _;  
}
```

```
constructor(Vault _vault) {  
    vault = _vault;  
}
```

```
function deposit(uint256 amount) external createActor {  
    depositCalls ++;
```

```
    amount = bound(amount, 0.01 ether, 10 ether);
```

```
    vm.deal(msg.sender, amount);  
    vm.prank(msg.sender);  
    vault.deposit{value: amount}();
```

```
    depositSuccess ++;  
}
```

```
function withdraw(uint256 amount) external createActor {  
    withdrawCalls ++;
```

```
    uint256 balance = vault.balances(msg.sender);  
    if (balance == 0) return;
```

```
    amount = bound(amount, 1, balance);
```

```
    vm.prank(msg.sender);  
    vault.withdraw(amount);
```

```
    withdrawSuccess ++;  
}
```

```
function getActors() external view returns (address[] memory) {  
    return actors;  
}
```

```
function sumOfAllBalances() external view returns (uint256) {  
    uint256 sum = 0;  
    for (uint256 i = 0; i < actors.length; i++) {  
        sum += vault.balances(actors[i]);  
    }  
    return sum;  
}
```

```
function callSummary() external view {  
    console.log("Deposit calls:", depositCalls);  
    console.log("Deposit successes:", depositSuccess);  
    console.log("Withdraw calls:", withdrawCalls);  
    console.log("Withdraw successes:", withdrawSuccess);  
    console.log("Total actors:", actors.length);  
}  
}
```

Complex Protocol Invariants

```
contract DeFiProtocolInvariantTest is Test {  
    LendingPool public pool;  
    Token public collateralToken;  
    Token public debtToken;  
    ProtocolHandler public handler;  
  
    function setUp() public {  
        collateralToken = new Token("Collateral", "COL", 0);  
        debtToken = new Token("Debt", "DEBT", 0);  
  
        pool = new LendingPool(  
            address(collateralToken),  
            address(debtToken)  
        );  
  
        debtToken.mint(address(pool), 1000000 ether);  
    }  
}
```

```
handler = new ProtocolHandler(pool, collateralToken, debtToken);

targetContract(address(handler));
}
```

```
function invariant_TotalBorrowedNeverExceedsLiquidity() public
view {
    assertLe(
        pool.totalBorrowed(),
        debtToken.balanceOf(address(pool)) + pool.totalBorrowed(),
        "Protocol is overleveraged"
    );
}
```

```
function invariant_CollateralRatioMaintained() public view {
    address[] memory borrowers = handler.getBorrowers();
```

```
    for (uint256 i = 0; i < borrowers.length; i++) {
        uint256 collateral = pool.collateralOf(borrowers[i]);
        uint256 debt = pool.debtOf(borrowers[i]);
```

```
        if (debt > 0) {
            uint256 ratio = (collateral * 100) / debt;
            assertGe(ratio, pool.minCollateralRatio(), "Undercollateralized
            position");
        }
    }
}
```

```
function invariant_NoOrphanedCollateral() public view {
    address[] memory depositors = handler.getDepositors();
```

```
    for (uint256 i = 0; i < depositors.length; i++) {
        uint256 collateral = pool.collateralOf(depositors[i]);
        uint256 debt = pool.debtOf(depositors[i]);
```

```
        if (collateral > 0) {
            assertTrue(
                debt > 0 || pool.canWithdrawCollateral(depositors[i]),
```

"Orphaned collateral"

```
);  
}  
}  
}
```

```
function invariant_ProtocolSolvency() public view {  
    uint256 totalCollateralValue = pool.totalCollateralValue();  
    uint256 totalDebt = pool.totalBorrowed();
```

```
    assertGe(  
        totalCollateralValue,  
        totalDebt,  
        "Protocol insolvent"  
    );  
}  
}
```

```
contract ProtocolHandler is Test {  
    LendingPool public pool;  
    Token public collateralToken;  
    Token public debtToken;
```

```
    address[] public depositors;  
    address[] public borrowers;  
    mapping(address => bool) public isDepositor;  
    mapping(address => bool) public isBorrower;
```

```
    constructor(  
        LendingPool _pool,  
        Token _collateralToken,  
        Token _debtToken  
    ) {  
        pool = _pool;  
        collateralToken = _collateralToken;  
        debtToken = _debtToken;  
    }
```

```
    function depositCollateral(uint256 amount) external {  
        amount = bound(amount, 1 ether, 100 ether);
```

```
if (!isDepositor[msg.sender]) {  
    depositors.push(msg.sender);  
    isDepositor[msg.sender] = true;  
}
```

```
collateralToken.mint(msg.sender, amount);
```

```
vm.startPrank(msg.sender);  
collateralToken.approve(address(pool), amount);  
pool.depositCollateral(amount);  
vm.stopPrank();  
}
```

```
function borrow(uint256 amount) external {  
    uint256 maxBorrow = pool.maxBorrowable(msg.sender);  
    if (maxBorrow == 0) return;
```

```
    amount = bound(amount, 1, maxBorrow);
```

```
    if (!isBorrower[msg.sender]) {  
        borrowers.push(msg.sender);  
        isBorrower[msg.sender] = true;  
    }
```

```
    vm.prank(msg.sender);  
    pool.borrow(amount);  
}
```

```
function repay(uint256 amount) external {  
    uint256 debt = pool.debtOf(msg.sender);  
    if (debt == 0) return;
```

```
    amount = bound(amount, 1, debt);
```

```
    debtToken.mint(msg.sender, amount);
```

```
    vm.startPrank(msg.sender);  
    debtToken.approve(address(pool), amount);  
    pool.repay(amount);
```



```

vm.stopPrank();
}

function withdrawCollateral(uint256 amount) external {
    uint256 withdrawable = pool.withdrawableCollateral(msg.sender);
    if (withdrawable == 0) return;

    amount = bound(amount, 1, withdrawable);

    vm.prank(msg.sender);
    pool.withdrawCollateral(amount);
}

function getDepositors() external view returns (address[] memory)
{
    return depositors;
}

function getBorrowers() external view returns (address[] memory)
{
    return borrowers;
}
}

```

SECTION 6: FORK TESTING

Testing Against Mainnet State

Fork testing runs your tests against actual mainnet data:

```

contract ForkTest is Test {
    uint256 mainnetFork;

    IERC20 public constant WETH =
    IERC20(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
    IERC20 public constant USDC =
    IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    IUniswapV2Router public constant ROUTER =
    IUniswapV2Router(0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D);
}

```

```
function setUp() public {
    mainnetFork =
    vm.createFork(vm.envString("MAINNET_RPC_URL"));
    vm.selectFork(mainnetFork);
}
```

```
function test_ForkBlockNumber() public view {
    assertGt(block.number, 15000000);
}
```

```
function test_UniswapSwap() public {
    address user = makeAddr("user");
    deal(address(WETH), user, 10 ether);
```

```
    vm.startPrank(user);
    WETH.approve(address(ROUTER), 10 ether);
```

```
    address[] memory path = new address[](2);
    path[0] = address(WETH);
    path[1] = address(USDC);
```

```
    uint256 usdcBefore = USDC.balanceOf(user);
```

```
    ROUTER.swapExactTokensForTokens(
        1 ether,
        0,
        path,
        user,
        block.timestamp + 1
    );
```

```
    uint256 usdcAfter = USDC.balanceOf(user);
    assertGt(usdcAfter, usdcBefore);
```

```
    vm.stopPrank();
}
}
```

Testing Protocol Integrations

```

contract IntegrationForkTest is Test {
    uint256 mainnetFork;

    MyProtocol public protocol;

    IERC20 public constant DAI =
IERC20(0x6B175474E89094C44Da98b954EesdfcdFB849);
    IAaveLendingPool public constant AAVE =
IAaveLendingPool(0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9);

    function setUp() public {
        mainnetFork =
vm.createFork(vm.envString("MAINNET_RPC_URL"), 18000000);
        vm.selectFork(mainnetFork);

        protocol = new MyProtocol(address(AAVE), address(DAI));
    }

    function test_IntegrationWithAave() public {
        address user = makeAddr("user");
        deal(address(DAI), user, 10000 ether);

        vm.startPrank(user);
        DAI.approve(address(protocol), 10000 ether);

        protocol.depositToAave(5000 ether);

        uint256 aTokenBalance = protocol.getATokenBalance(user);
        assertGt(aTokenBalance, 0);

        vm.stopPrank();
    }

    function test_FlashLoanIntegration() public {
        address[] memory assets = new address[](1);
        assets[0] = address(DAI);

        uint256[] memory amounts = new uint256[](1);
        amounts[0] = 1000000 ether;
    }
}

```

```

vm.expectEmit(true, true, false, false);
emit FlashLoanExecuted(assets[0], amounts[0]);

protocol.executeFlashLoan(assets, amounts);
}

event FlashLoanExecuted(address asset, uint256 amount);
}

```

Multi-Chain Fork Testing

```

contract MultiChainForkTest is Test {
    uint256 mainnetFork;
    uint256 arbitrumFork;
    uint256 optimismFork;

    function setUp() public {
        mainnetFork =
        vm.createFork(vm.envString("MAINNET_RPC_URL"));
        arbitrumFork =
        vm.createFork(vm.envString("ARBITRUM_RPC_URL"));
        optimismFork =
        vm.createFork(vm.envString("OPTIMISM_RPC_URL"));
    }

    function test_MainnetState() public {
        vm.selectFork(mainnetFork);
        assertEq(block.chainid, 1);
    }

    function test_ArbitrumState() public {
        vm.selectFork(arbitrumFork);
        assertEq(block.chainid, 42161);
    }

    function test_OptimismState() public {
        vm.selectFork(optimismFork);
        assertEq(block.chainid, 10);
    }
}

```

```

function test_CrossChainBehavior() public {
    vm.selectFork(mainnetFork);
    uint256 mainnetBalance = address(0x123).balance;

    vm.selectFork(arbitrumFork);
    uint256 arbitrumBalance = address(0x123).balance;

    console.log("Mainnet balance:", mainnetBalance);
    console.log("Arbitrum balance:", arbitrumBalance);
}
}

```

SECTION 7: GAS PROFILING AND OPTIMIZATION

Gas Snapshot Testing

Run gas snapshots:

```
forge snapshot
```

Compare gas changes:

```
forge snapshot --diff
```

Gas profiling in tests:

```

contract GasOptimizationTest is Test {
    Vault public vault;
    OptimizedVault public optimizedVault;

    function setUp() public {
        vault = new Vault();
        optimizedVault = new OptimizedVault();

        vm.deal(address(this), 1000 ether);
    }

    function test_DepositGasComparison() public {

```

```

uint256 gasBefore = gasleft();
vault.deposit{value: 1 ether}();
uint256 gasUsed = gasBefore - gasleft();
console.log("Original deposit gas:", gasUsed);

```

```

gasBefore = gasleft();
optimizedVault.deposit{value: 1 ether}();
gasUsed = gasBefore - gasleft();
console.log("Optimized deposit gas:", gasUsed);
}

```

```

function test_BatchDepositGas() public {
uint256 gasBefore = gasleft();
for (uint256 i = 0; i < 10; i++) {
vault.deposit{value: 0.1 ether}();
}
uint256 gasUsed = gasBefore - gasleft();
console.log("10 individual deposits gas:", gasUsed);

```

```

gasBefore = gasleft();
optimizedVault.batchDeposit{value: 1 ether}(10);
gasUsed = gasBefore - gasleft();
console.log("Batch deposit gas:", gasUsed);
}
}

```

Gas Limits and DoS Testing

```

contract GasLimitTest is Test {
UnboundedLoop public vulnerable;
BoundedLoop public safe;

function setUp() public {
vulnerable = new UnboundedLoop();
safe = new BoundedLoop();
}

```

```

function test_UnboundedLoopDoS() public {
for (uint256 i = 0; i < 1000; i++) {
vulnerable.addUser(address(uint160(i)));

```

```

}

uint256 gasBefore = gasleft();
try vulnerable.distributeToAll{gas: 300000000}(1 ether) {
} catch {
console.log("Distribution failed at 1000 users");
}
uint256 gasUsed = gasBefore - gasleft();
console.log("Gas used:", gasUsed);
}

function test_BoundedLoopSafe() public {
for (uint256 i = 0; i < 1000; i++) {
safe.addUser(address(uint160(i)));
}

uint256 distributed = 0;
uint256 batchSize = 100;

while (distributed < 1000) {
safe.distributeInBatches(1 ether, batchSize);
distributed += batchSize;
}

assertEq(distributed, 1000);
}
}

```

SECTION 8: COMPREHENSIVE TEST SUITE

Putting It All Together

```

contract ComprehensiveVaultTest is Test {
Vault public vault;

address public owner;
address public admin;
address public user1;
address public user2;

```

```
address public attacker;
```

```
event Deposited(address indexed user, uint256 amount);  
event Withdrawn(address indexed user, uint256 amount);
```

```
function setUp() public {  
    owner = makeAddr("owner");  
    admin = makeAddr("admin");  
    user1 = makeAddr("user1");  
    user2 = makeAddr("user2");  
    attacker = makeAddr("attacker");
```

```
    vm.deal(owner, 100 ether);  
    vm.deal(admin, 100 ether);  
    vm.deal(user1, 100 ether);  
    vm.deal(user2, 100 ether);  
    vm.deal(attacker, 100 ether);
```

```
    vm.prank(owner);  
    vault = new Vault();  
}
```

```
function test_Deployment() public view {  
    assertEq(vault.owner(), owner);  
    assertEq(address(vault).balance, 0);  
    assertEq(vault.totalDeposits(), 0);  
}
```

```
function test_Deposit_Success() public {  
    vm.prank(user1);  
    vm.expectEmit(true, false, false, true);  
    emit Deposited(user1, 5 ether);  
    vault.deposit{value: 5 ether}();
```

```
    assertEq(vault.balances(user1), 5 ether);  
    assertEq(address(vault).balance, 5 ether);  
    assertEq(vault.totalDeposits(), 5 ether);  
}
```

```
function test_Deposit_MultipleDeposits() public {
```



```
vm.startPrank(user1);
vault.deposit{value: 1 ether}();
vault.deposit{value: 2 ether}();
vault.deposit{value: 3 ether}();
vm.stopPrank();

assertEq(vault.balances(user1), 6 ether);
}
```

```
function test_Deposit_MultipleUsers() public {
vm.prank(user1);
vault.deposit{value: 5 ether}();
```

```
vm.prank(user2);
vault.deposit{value: 3 ether}();
```

```
assertEq(vault.balances(user1), 5 ether);
assertEq(vault.balances(user2), 3 ether);
assertEq(vault.totalDeposits(), 8 ether);
}
```

```
function test_Deposit_RevertOnZero() public {
vm.prank(user1);
vm.expectRevert("Cannot deposit zero");
vault.deposit{value: 0}();
}
```

```
function test-Withdraw_Success() public {
vm.startPrank(user1);
vault.deposit{value: 10 ether}();
```

```
uint256 balanceBefore = user1.balance;
```

```
vm.expectEmit(true, false, false, true);
emit Withdrawn(user1, 5 ether);
vault.withdraw(5 ether);
```

```
assertEq(user1.balance, balanceBefore + 5 ether);
assertEq(vault.balances(user1), 5 ether);
vm.stopPrank();
```

```
}
```

```
function test_Withdraw_Full() public {  
    vm.startPrank(user1);  
    vault.deposit{value: 10 ether}();  
    vault.withdraw(10 ether);  
    vm.stopPrank();
```

```
    assertEq(vault.balances(user1), 0);  
}
```

```
function test_Withdraw_RevertInsufficientBalance() public {  
    vm.startPrank(user1);  
    vault.deposit{value: 5 ether}();
```

```
    vm.expectRevert("Insufficient balance");  
    vault.withdraw(6 ether);  
    vm.stopPrank();  
}
```

```
function test_Withdraw_RevertZeroBalance() public {  
    vm.prank(user1);  
    vm.expectRevert("Insufficient balance");  
    vault.withdraw(1 ether);  
}
```

```
function test_Security_ReentrancyProtection() public {  
    vm.prank(owner);  
    vault.deposit{value: 50 ether}();
```

```
    ReentrancyAttacker attackContract = new  
    ReentrancyAttacker(address(vault));  
    vm.deal(address(attackContract), 5 ether);
```

```
    uint256 vaultBalanceBefore = address(vault).balance;
```

```
    vm.expectRevert();  
    attackContract.attack{value: 1 ether}();
```

```
    assertEq(address(vault).balance, vaultBalanceBefore);
```

```
}
```

```
function test_Security_AccessControl() public {  
    vm.prank(attacker);  
    vm.expectRevert("Not owner");  
    vault.emergencyWithdraw();  
}
```

```
function testFuzz_DepositWithdraw(uint256 depositAmount,  
uint256 withdrawAmount) public {  
    depositAmount = bound(depositAmount, 0.01 ether, 50 ether);
```

```
    vm.startPrank(user1);  
    vault.deposit{value: depositAmount}();
```

```
    withdrawAmount = bound(withdrawAmount, 0, depositAmount);  
    vault.withdraw(withdrawAmount);
```

```
    assertEq(vault.balances(user1), depositAmount -  
withdrawAmount);  
    vm.stopPrank();  
}
```

```
function testFuzz_MultipleOperations(uint256 seed) public {  
    uint256 totalDeposited = 0;  
    uint256 totalWithdrawn = 0;
```

```
    for (uint256 i = 0; i < 5; i++) {  
        uint256 amount = bound(  
            uint256(keccak256(abi.encode(seed, i))),  
            0.1 ether,  
            5 ether  
        );
```

```
        vm.prank(user1);  
        vault.deposit{value: amount}();  
        totalDeposited += amount;  
    }
```

```
    for (uint256 i = 0; i < 3; i++) {
```

```

uint256 balance = vault.balances(user1);
if (balance == 0) break;

uint256 amount = bound(
uint256(keccak256(abi.encode(seed, "withdraw", i))),
1,
balance
);

vm.prank(user1);
vault.withdraw(amount);
totalWithdrawn += amount;
}

assertEq(vault.balances(user1), totalDeposited - totalWithdrawn);
}
}

```

Running the Complete Test Suite

Run all tests:

```
forge test
```

Run with verbosity:

```
forge test -vvvv
```

Run specific test:

```
forge test --match-test test_Deposit_Success
```

Run with gas report:

```
forge test --gas-report
```

Run with coverage:

```
forge coverage
```

This chapter covered runtime analysis from basic unit tests through advanced invariant and fork testing. The next chapter explores formal verification techniques that provide mathematical guarantees of correctness.

BOOK 7: SECURITY AND AUDITING

Chapter 5: Formal Verification

Testing can only prove the presence of bugs, never their absence. You cannot test every possible input. You cannot test every possible state. Formal verification mathematically proves that code satisfies its specification for all possible inputs. This chapter teaches you when and how to apply formal methods to smart contracts.

SECTION 1: UNDERSTANDING FORMAL VERIFICATION

What Formal Verification Proves

Formal verification uses mathematics to prove properties about code. Instead of testing with specific inputs, you prove statements like:

"For ALL possible inputs and ALL possible states, this property holds."

Consider a simple property: "Users cannot withdraw more than their balance."

Testing approach:

Test with balance = 100, withdraw = 50. Works.

Test with balance = 100, withdraw = 100. Works.

Test with balance = 100, withdraw = 101. Reverts. Good.

Test with balance = 0, withdraw = 1. Reverts. Good.

But did you test with balance = `type(uint256).max`? Did you test when two users interact simultaneously? Did you test after 1000 deposits and withdrawals?

Formal verification approach:

Prove mathematically that for ANY balance value and ANY withdrawal amount, if $\text{withdrawal} > \text{balance}$, the function reverts.

This proof covers ALL cases. Not just the ones you thought to test.

Types of Properties

Safety Properties:

"Bad things never happen."

Examples:

- Users never lose funds
- Total supply never exceeds maximum
- Only owner can call admin functions

Liveness Properties:

"Good things eventually happen."

Examples:

- Withdrawals eventually complete
- Locked funds eventually unlock
- Auctions eventually settle

Invariants:

Properties that hold at every observable state.

Examples:

- $\text{sum}(\text{balances}) == \text{totalSupply}$
- $\text{collateral} \geq \text{debt} * \text{ratio}$
- paused implies no state changes

Limitations

Formal verification proves code matches specification. It does not:

1. Prove specification is correct

If your specification is wrong, the proof is useless. You formally verified the wrong thing.

2. Prove economic security

Mathematical proofs do not capture market dynamics, game theory, or incentive structures.

3. Handle external dependencies

Proofs assume external contracts behave as specified. Malicious externals break assumptions.

4. Scale infinitely

Complex contracts may be computationally infeasible to fully verify.

SECTION 2: SPECIFICATION LANGUAGES

Writing Formal Specifications

Specifications describe what code SHOULD do. Solidity comments are informal. Formal specifications use precise mathematical notation.

Natural Language to Formal Specification:

Natural: "Users can only withdraw their own balance"

Formal: $\forall \text{user, amount: } \text{withdraw}(\text{amount}) \text{ succeeds} \implies \text{balances}[\text{user}] \geq \text{amount}$

Natural: "Total supply never changes after minting phase"

Formal: $\text{mintingFinished} \implies \text{totalSupply} = \text{totalSupply_old}$

Natural: "Only owner can pause"

Formal: $\text{pause}() \text{ caller} \neq \text{owner} \implies \text{revert}$

Solidity NatSpec for Verification

NatSpec comments can express properties:

```
contract Vault {  
    mapping(address => uint256) public balances;  
    uint256 public totalDeposits;
```

```
    function deposit() external payable {  
        balances[msg.sender] += msg.value;
```

```

totalDeposits += msg.value;
}

function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient");

    balances[msg.sender] -= amount;
    totalDeposits -= amount;

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
}

```

Writing Verification Conditions

For withdraw function, verification conditions might be:

1. Precondition: $\text{balances}[\text{msg.sender}] \geq \text{amount}$
2. Postcondition: $\text{balances}'[\text{msg.sender}] = \text{balances}[\text{msg.sender}] - \text{amount}$
3. Postcondition: $\text{totalDeposits}' = \text{totalDeposits} - \text{amount}$
4. Invariant: $\text{address(this).balance} \geq \text{totalDeposits}$

Where $\text{balances}'$ denotes the value after execution.

SECTION 3: CERTORA PROVER

Introduction to Certora

Certora is the most widely used formal verification tool for smart contracts. It uses its own specification language called CVL (Certora Verification Language).

Installation and Setup:

```
pip install certora-cli
```

Configuration file `certora.conf`:


```
{
  "files": [
    "contracts/Vault.sol"
  ],
  "verify": "Vault:specs/Vault.spec",
  "solc": "solc",
  "msg": "Vault verification"
}
```

CVL Specification Basics

```
methods {
  function balances(address) external returns (uint256) envfree;
  function totalDeposits() external returns (uint256) envfree;
  function deposit() external;
  function withdraw(uint256) external;
}
```

```
rule withdrawReducesBalance {
  env e;
  uint256 amount;

  uint256 balanceBefore = balances(e.msg.sender);

  withdraw(e, amount);

  uint256 balanceAfter = balances(e.msg.sender);

  assert balanceAfter == balanceBefore - amount;
}
```

```
rule cannotWithdrawMoreThanBalance {
  env e;
  uint256 amount;

  uint256 balance = balances(e.msg.sender);

  require amount > balance;
```

```
withdraw@withrevert(e, amount);
```

```
assert lastReverted;  
}
```

```
invariant totalMatchesSum()  
totalDeposits() == sumOfBalances()
```

Complete Certora Specification

```
methods {  
  function balances(address) external returns (uint256) envfree;  
  function totalDeposits() external returns (uint256) envfree;  
  function owner() external returns (address) envfree;  
  function paused() external returns (bool) envfree;  
  function deposit() external;  
  function withdraw(uint256) external;  
  function pause() external;  
  function unpause() external;  
  function emergencyWithdraw() external;  
}
```

```
definition isOwner(address a) returns bool =  
  a == owner();
```

```
ghost mapping(address => uint256) ghostBalances {  
  init_state axiom forall address a. ghostBalances[a] == 0;  
}
```

```
ghost uint256 sumOfGhostBalances {  
  init_state axiom sumOfGhostBalances == 0;  
}
```

```
hook Sstore balances[KEY address a] uint256 newValue (uint256  
oldValue) STORAGE {  
  sumOfGhostBalances = sumOfGhostBalances + newValue -  
oldValue;  
  ghostBalances[a] = newValue;  
}
```

```
hook Sload uint256 value balances[KEY address a] STORAGE {  
  require ghostBalances[a] == value;  
}
```

```
invariant solvency()  
  nativeBalances[currentContract] >= totalDeposits()  
  {  
    preserved with (env e) {  
      require e.msg.sender != currentContract;  
    }  
  }
```

```
invariant totalMatchesBalances()  
  sumOfGhostBalances == totalDeposits()
```

```
rule depositIncreasesBalance {  
  env e;  
  require e.msg.value > 0;  
  
  uint256 balanceBefore = balances(e.msg.sender);  
  
  deposit(e);  
  
  uint256 balanceAfter = balances(e.msg.sender);  
  
  assert balanceAfter == balanceBefore + e.msg.value;  
}
```

```
rule withdrawDecreasesBalance {  
  env e;  
  uint256 amount;  
  
  uint256 balanceBefore = balances(e.msg.sender);  
  require balanceBefore >= amount;  
  
  withdraw(e, amount);  
  
  uint256 balanceAfter = balances(e.msg.sender);  
  
  assert balanceAfter == balanceBefore - amount;
```

```

}

rule onlyOwnerCanPause {
  env e;

  require !isOwner(e.msg.sender);

  pause@withrevert(e);

  assert lastReverted;
}

rule pausedContractRejectsDeposits {
  env e;

  require paused();

  deposit@withrevert(e);

  assert lastReverted;
}

rule noReentrancyVulnerability {
  env e;
  uint256 amount;

  storage initialStorage = lastStorage;

  withdraw(e, amount);

  uint256 balanceAfter = balances(e.msg.sender);

  assert balanceAfter == ghostBalances[e.msg.sender];
}

```

SECTION 4: SYMBOLIC EXECUTION WITH HALMOS

Understanding Halmos

Halmos performs symbolic execution on Foundry tests. It explores all possible execution paths mathematically.

Installation:

```
pip install halmos
```

Writing Halmos Tests

Halmos tests look like Foundry tests but use symbolic values:

```
import {Test} from "forge-std/Test.sol";
import {Vault} from "../src/Vault.sol";

contract VaultSymbolicTest is Test {
    Vault vault;

    function setUp() public {
        vault = new Vault();
    }

    function check_withdrawNeverExceedsBalance(
        address user,
        uint256 depositAmount,
        uint256 withdrawAmount
    ) public {
        vm.assume(depositAmount > 0);
        vm.assume(depositAmount <= 100 ether);
        vm.assume(user != address(0));
        vm.assume(user != address(vault));

        vm.deal(user, depositAmount);
        vm.prank(user);
        vault.deposit{value: depositAmount}();

        uint256 balance = vault.balances(user);

        if (withdrawAmount > balance) {
            vm.prank(user);
            vm.expectRevert();
        }
    }
}
```

```

vault.withdraw(withdrawAmount);
} else {
vm.prank(user);
vault.withdraw(withdrawAmount);

uint256 newBalance = vault.balances(user);
assert(newBalance == balance - withdrawAmount);
}
}

```

```

function check_depositAlwaysIncreasesBalance(
address user,
uint256 amount
) public {
vm.assume(amount > 0);
vm.assume(amount <= 100 ether);
vm.assume(user != address(0));
vm.assume(user != address(vault));

```

```

uint256 balanceBefore = vault.balances(user);

```

```

vm.deal(user, amount);
vm.prank(user);
vault.deposit{value: amount}();

```

```

uint256 balanceAfter = vault.balances(user);

```

```

assert(balanceAfter == balanceBefore + amount);
}

```

```

function check_invariant_solveny(
address user1,
address user2,
uint256 amount1,
uint256 amount2,
uint256 withdraw1
) public {
vm.assume(user1 != address(0) && user2 != address(0));
vm.assume(user1 != user2);
vm.assume(user1 != address(vault) && user2 != address(vault));

```

```

vm.assume(amount1 > 0 && amount2 > 0);
vm.assume(amount1 <= 50 ether && amount2 <= 50 ether);

vm.deal(user1, amount1);
vm.prank(user1);
vault.deposit{value: amount1}();

vm.deal(user2, amount2);
vm.prank(user2);
vault.deposit{value: amount2}();

uint256 balance1 = vault.balances(user1);
withdraw1 = withdraw1 % (balance1 + 1);

if (withdraw1 > 0 && withdraw1 <= balance1) {
    vm.prank(user1);
    vault.withdraw(withdraw1);
}

uint256 totalDeposits = vault.totalDeposits();
uint256 contractBalance = address(vault).balance;

assert(contractBalance >= totalDeposits);
}
}

```

Running Halmos:

```
halmos --contract VaultSymbolicTest
```

SECTION 5: SMT SOLVERS

Understanding SMT

SMT (Satisfiability Modulo Theories) solvers are the engines behind formal verification. They solve mathematical constraints to find counterexamples or prove properties.

SMT Solver Integration

```
from z3 import Solver, Int, If, And, Or, Not, sat, unsat
```

```
class SmartContractVerifier:
```

```
    def __init__(self):  
        self.solver = Solver()
```

```
    def verify_withdrawal_safety(self):  
        balance = Int("balance")  
        withdraw_amount = Int("withdraw_amount")  
        new_balance = Int("new_balance")
```

```
        self.solver.push()
```

```
        self.solver.add(balance >= 0)  
        self.solver.add(withdraw_amount >= 0)  
        self.solver.add(withdraw_amount <= balance)
```

```
        self.solver.add(new_balance == balance - withdraw_amount)
```

```
        self.solver.add(new_balance < 0)
```

```
        result = self.solver.check()
```

```
        self.solver.pop()
```

```
        if result == unsat:  
            return True, "Withdrawal is safe: balance can never go negative"  
        else:  
            model = self.solver.model()  
            return False, f"Counterexample found: {model}"
```

```
    def verify_no_overflow(self, bit_width = 256):  
        max_value = 2**bit_width - 1
```

```
        a = Int("a")  
        b = Int("b")  
        result = Int("result")
```



```

self.solver.push()

self.solver.add(a >= 0, a <= max_value)
self.solver.add(b >= 0, b <= max_value)

self.solver.add(result == a + b)

self.solver.add(result > max_value)

check_result = self.solver.check()

self.solver.pop()

if check_result == sat:
    model = self.solver.model()
    return False, f"Overflow possible: a = {model[a]}, b = {model[b]}"
else:
    return True, "No overflow possible within constraints"

def verify_access_control(self):
    caller = Int("caller")
    owner = Int("owner")
    admin = Int("admin")
    can_execute = Int("can_execute")

    self.solver.push()

    self.solver.add(owner == 1)
    self.solver.add(admin == 2)

    self.solver.add(
        can_execute == If(
            Or(caller == owner, caller == admin),
            1,
            0
        )
    )

    self.solver.add(caller != owner)

```

```

self.solver.add(caller != admin)
self.solver.add(can_execute == 1)

result = self.solver.check()

self.solver.pop()

if result == unsat:
    return True, "Access control is secure"
else:
    model = self.solver.model()
    return False, f"Access control bypassed: caller = {model[caller]}"

```

```

verifier = SmartContractVerifier()

```

```

safe, message = verifier.verify_withdrawal_safety()
print(f"Withdrawal safety: {message}")

```

```

safe, message = verifier.verify_no_overflow()
print(f"Overflow check: {message}")

```

```

safe, message = verifier.verify_access_control()
print(f"Access control: {message}")

```

Complex Property Verification

```

class DeFiPropertyVerifier:

```

```

    def __init__(self):
        from z3 import Solver, Real, Int, And, Or, Implies
        self.Solver = Solver
        self.Real = Real
        self.Int = Int
        self.And = And
        self.Or = Or
        self.Implies = Implies

```

```

    def verify_amm_constant_product(self):
        solver = self.Solver()

```

```
reserve_x_before = self.Real("reserve_x_before")
reserve_y_before = self.Real("reserve_y_before")
reserve_x_after = self.Real("reserve_x_after")
reserve_y_after = self.Real("reserve_y_after")
```

```
amount_in = self.Real("amount_in")
amount_out = self.Real("amount_out")
```

```
k_before = self.Real("k_before")
k_after = self.Real("k_after")
```

```
solver.add(reserve_x_before > 0)
solver.add(reserve_y_before > 0)
solver.add(amount_in > 0)
solver.add(amount_out > 0)
```

```
solver.add(k_before == reserve_x_before * reserve_y_before)
```

```
solver.add(reserve_x_after == reserve_x_before + amount_in)
solver.add(reserve_y_after == reserve_y_before - amount_out)
```

```
solver.add(k_after == reserve_x_after * reserve_y_after)
```

```
solver.add(k_after >= k_before)
```

```
solver.add(amount_out > reserve_y_before)
```

```
from z3 import sat, unsat
result = solver.check()
```

```
if result == unsat:
    return True, "AMM invariant holds: cannot drain reserves"
else:
    model = solver.model()
    return False, f"Invariant violated: {model}"
```

```
def verify_liquidation_safety(self):
    solver = self.Solver()
```

```
collateral = self.Real("collateral")
debt = self.Real("debt")
collateral_ratio = self.Real("collateral_ratio")
liquidation_threshold = self.Real("liquidation_threshold")
```

```
price_before = self.Real("price_before")
price_after = self.Real("price_after")
```

```
is_liquidatable = self.Int("is_liquidatable")
```

```
solver.add(collateral > 0)
solver.add(debt > 0)
solver.add(price_before > 0)
solver.add(price_after > 0)
solver.add(liquidation_threshold == 150)
```

```
solver.add(collateral_ratio == (collateral * price_after * 100) /
debt)
```

```
from z3 import If
solver.add(
    is_liquidatable == If(
        collateral_ratio < liquidation_threshold,
        1,
        0
    )
)
```

```
solver.add(is_liquidatable == 1)
solver.add(collateral_ratio >= liquidation_threshold)
```

```
from z3 import sat, unsat
result = solver.check()
```

```
if result == unsat:
    return True, "Liquidation logic is consistent"
else:
    return False, "Liquidation logic has inconsistency"
```

SECTION 6: PROPERTY-BASED VERIFICATION

Defining Properties

Properties express what your contract MUST do:

```
from dataclasses import dataclass
from typing import List, Callable, Any
from enum import Enum
```

```
class PropertyType(Enum):
    INVARIANT = "invariant"
    PRECONDITION = "precondition"
    POSTCONDITION = "postcondition"
    SAFETY = "safety"
    LIVENESS = "liveness"
```

```
@dataclass
class Property:
    name: str
    property_type: PropertyType
    description: str
    formula: str
    check_function: Callable[..., bool]
```

```
class PropertyRegistry:

    def __init__(self):
        self.properties: List[Property] = []

    def add_invariant(
        self,
        name: str,
        description: str,
        formula: str,
        check_function: Callable
    ):
```

```

prop = Property(
    name = name,
    property_type = PropertyType.INVARIANT,
    description = description,
    formula = formula,
    check_function = check_function
)
self.properties.append(prop)

def add_safety(
    self,
    name: str,
    description: str,
    formula: str,
    check_function: Callable
):
    prop = Property(
        name = name,
        property_type = PropertyType.SAFETY,
        description = description,
        formula = formula,
        check_function = check_function
    )
    self.properties.append(prop)

def verify_all(self, contract_state: Any) -> List[tuple]:
    results = []

    for prop in self.properties:
        try:
            holds = prop.check_function(contract_state)
            results.append((prop.name, holds, None))
        except Exception as e:
            results.append((prop.name, False, str(e)))

    return results

registry = PropertyRegistry()

```

```

registry.add_invariant(
    name="solvency",
    description="Contract balance covers all deposits",
    formula="balance(contract) >= sum(balances)",
    check_function=lambda state: state.contract_balance >=
sum(state.user_balances.values())
)

```

```

registry.add_invariant(
    name="total_consistency",
    description="Total deposits equals sum of individual balances",
    formula="totalDeposits == sum(balances[i] for all i)",
    check_function=lambda state: state.total_deposits ==
sum(state.user_balances.values())
)

```

```

registry.add_safety(
    name="no_negative_balance",
    description="No user can have negative balance",
    formula="forall user: balances[user] >= 0",
    check_function=lambda state: all(b >= 0 for b in
state.user_balances.values())
)

```

Automated Property Generation

```

class PropertyGenerator:

```

```

    def __init__(self, contract_abi: List[dict]):
        self.abi = contract_abi
        self.properties = []

```

```

    def generate_view_consistency_properties(self):
        view_functions = [
            f for f in self.abi
            if f.get("type") == "function"
            and f.get("stateMutability") in ["view", "pure"]
        ]

```

```

        for func in view_functions:

```

```

name = func["name"]
self.properties.append({
    "name": f'{name}_deterministic',
    "description": f'{name} returns same value when called twice',
    "type": "invariant"
})

```

```

def generate_access_control_properties(self):
    state_changing = [
        f for f in self.abi
        if f.get("type") == "function"
        and f.get("stateMutability") not in ["view", "pure"]
    ]

```

```

admin_keywords = ["owner", "admin", "pause", "emergency",
"upgrade", "set"]

```

```

for func in state_changing:
    name = func["name"]
    if any(keyword in name.lower() for keyword in admin_keywords):
        self.properties.append({
            "name": f'{name}_access_controlled',
            "description": f'{name} should have access control",
            "type": "safety"
        })

```

```

def generate_balance_properties(self):
    has_balance = any(
        f.get("name") == "balanceOf" or f.get("name") == "balances"
        for f in self.abi
    )

```

```

if has_balance:
    self.properties.append({
        "name": "total_supply_consistency",
        "description": "Sum of balances equals total supply",
        "type": "invariant"
    })

```

```

self.properties.append({

```



```
"name": "no_balance_creation",
"description": "Transfers do not create tokens",
"type": "safety"
})
```

```
def generate_all(self) -> List[dict]:
    self.generate_view_consistency_properties()
    self.generate_access_control_properties()
    self.generate_balance_properties()
    return self.properties
```

SECTION 7: VERIFICATION WORKFLOW

Complete Verification Pipeline

```
from dataclasses import dataclass, field
from typing import List, Dict, Optional
from enum import Enum
import json
```

```
class VerificationStatus(Enum):
    PENDING = "pending"
    RUNNING = "running"
    PASSED = "passed"
    FAILED = "failed"
    TIMEOUT = "timeout"
    ERROR = "error"
```

```
@dataclass
class VerificationResult:
    property_name: str
    status: VerificationStatus
    counterexample: Optional[Dict] = None
    proof_time: float = 0.0
    details: str = ""
```

```

@dataclass
class VerificationReport:
    contract_name: str
    total_properties: int
    passed: int
    failed: int
    timeout: int
    results: List[VerificationResult] = field(default_factory=list)

    def summary(self) -> str:
        return f"""
Verification Report: {self.contract_name}

Total Properties: {self.total_properties}
Passed: {self.passed}
Failed: {self.failed}
Timeout: {self.timeout}

Pass Rate: {(self.passed / self.total_properties * 100):.1f}%
"""

```

```

class VerificationPipeline:

    def __init__(self, contract_path: str, spec_path: str):
        self.contract_path = contract_path
        self.spec_path = spec_path
        self.results: List[VerificationResult] = []

    def run_certora(self) -> List[VerificationResult]:
        import subprocess

        cmd = [
            "certoraRun",
            self.contract_path,
            "--verify",
            f'Contract:{self.spec_path}',
            "--json"
        ]

```

```

try:
    result = subprocess.run(
        cmd,
        capture_output=True,
        text=True,
        timeout=3600
    )

    if result.stdout:
        output = json.loads(result.stdout)
        return self._parse_certora_results(output)

```

```

except subprocess.TimeoutExpired:
    return [VerificationResult(
        property_name="all",
        status=VerificationStatus.TIMEOUT,
        details="Verification timed out after 1 hour"
    )]

except Exception as e:
    return [VerificationResult(
        property_name="all",
        status=VerificationStatus.ERROR,
        details=str(e)
    )]

```

```

return []

```

```

def _parse_certora_results(self, output: dict) ->
List[VerificationResult]:
    results = []

```

```

    for rule in output.get("rules", []):
        status = VerificationStatus.PASSED if rule.get("status") ==
"verified" else VerificationStatus.FAILED

```

```

    result = VerificationResult(
        property_name=rule.get("name", "unknown"),
        status=status,
        counterexample=rule.get("counterexample"),
        proof_time=rule.get("time", 0),

```

```

details = rule.get("message", "")
)
results.append(result)

return results

def run_halmos(self) -> List[VerificationResult]:
import subprocess

cmd = ["halmos", "--contract", self.contract_path]

try:
result = subprocess.run(
cmd,
capture_output=True,
text=True,
timeout=1800
)

return self._parse_halmos_results(result.stdout)

except subprocess.TimeoutExpired:
return [VerificationResult(
property_name="all",
status=VerificationStatus.TIMEOUT
)]
except Exception as e:
return [VerificationResult(
property_name="all",
status=VerificationStatus.ERROR,
details=str(e)
)]

def _parse_halmos_results(self, output: str) ->
List[VerificationResult]:
results = []

for line in output.split("\n"):
if "check_" in line:
if "PASS" in line:

```

```

name = line.split()[0]
results.append(VerificationResult(
    property_name = name,
    status = VerificationStatus.PASSED
))
elif "FAIL" in line:
    name = line.split()[0]
    results.append(VerificationResult(
        property_name = name,
        status = VerificationStatus.FAILED,
        details = line
    ))

return results

def generate_report(self) -> VerificationReport:
    passed = len([r for r in self.results if r.status ==
VerificationStatus.PASSED])
    failed = len([r for r in self.results if r.status ==
VerificationStatus.FAILED])
    timeout = len([r for r in self.results if r.status ==
VerificationStatus.TIMEOUT])

    return VerificationReport(
        contract_name = self.contract_path,
        total_properties = len(self.results),
        passed = passed,
        failed = failed,
        timeout = timeout,
        results = self.results
    )

```

SECTION 8: PRACTICAL VERIFICATION STRATEGIES

What to Verify

Not everything needs formal verification. Focus on:

1. Critical Financial Logic

- Token minting and burning
- Withdrawal functions
- Fee calculations
- Liquidation logic

2. Access Control

- Owner-only functions
- Role-based permissions
- Upgrade mechanisms

3. Invariants

- Conservation laws (tokens in == tokens out)
- Solvency conditions
- State machine transitions

4. Mathematical Properties

- No overflow in critical calculations
- Correct rounding behavior
- Price oracle bounds

Incremental Verification

Start simple, add complexity:

```
methods {
  function deposit() external;
  function withdraw(uint256) external;
  function balances(address) external returns (uint256) envfree;
}
```

```
rule basicWithdrawWorks {
  env e;
  uint256 amount;

  require balances(e.msg.sender) >= amount;

  withdraw(e, amount);

  assert balances(e.msg.sender) >= 0;
}
```

```

rule withdrawReducesBalance {
  env e;
  uint256 amount;

  uint256 before = balances(e.msg.sender);
  require before >= amount;

  withdraw(e, amount);

  assert balances(e.msg.sender) == before - amount;
}

```

```

rule withdrawCannotExceedBalance {
  env e;
  uint256 amount;

  uint256 balance = balances(e.msg.sender);
  require amount > balance;

  withdraw@withrevert(e, amount);

  assert lastReverted;
}

```

```

invariant solvency()
  nativeBalances[currentContract] >= totalDeposits()

```

Handling Verification Failures

When verification fails, you get a counterexample. Analyze it:

```

class CounterexampleAnalyzer:

  def __init__(self, counterexample: dict):
    self.ce = counterexample

  def analyze(self) -> dict:
    analysis = {
      "initial_state": self._extract_initial_state(),

```

```
"transactions": self._extract_transactions(),
"final_state": self._extract_final_state(),
"violated_property": self._extract_violation(),
"root_cause": self._identify_root_cause()
}
return analysis
```

```
def _extract_initial_state(self) -> dict:
return self.ce.get("initial_state", {})
```

```
def _extract_transactions(self) -> list:
return self.ce.get("transactions", [])
```

```
def _extract_final_state(self) -> dict:
return self.ce.get("final_state", {})
```

```
def _extract_violation(self) -> str:
return self.ce.get("assertion", "Unknown")
```

```
def _identify_root_cause(self) -> str:
txs = self._extract_transactions()
```

```
if len(txs) == 1:
return "Single transaction causes violation - check function logic"
```

```
tx_types = [tx.get("function", "") for tx in txs]
```

```
if len(set(tx_types)) == 1:
return f"Multiple {tx_types[0]} calls cause violation - check
reentrancy or accumulation"
```

```
return "Multi-step attack - analyze transaction sequence"
```

```
def suggest_fix(self) -> list:
suggestions = []
```

```
root_cause = self._identify_root_cause()
```

```
if "reentrancy" in root_cause.lower():
suggestions.append("Add ReentrancyGuard to affected functions")
```



```
suggestions.append("Apply Checks-Effects-Interactions pattern")
```

```
if "accumulation" in root_cause.lower():
```

```
suggestions.append("Add limits on repeated operations")
```

```
suggestions.append("Check for integer overflow in accumulation")
```

```
if "multi-step" in root_cause.lower():
```

```
suggestions.append("Review state machine transitions")
```

```
suggestions.append("Add invariant checks between operations")
```

```
return suggestions
```

This chapter covered formal verification from theory to practice. Formal methods provide the highest assurance but require significant investment. The next chapter covers common vulnerability patterns and how to prevent them systematically.

BOOK 7: SECURITY AND AUDITING

Chapter 6: Common Vulnerability Patterns

Attack patterns repeat. The same vulnerabilities appear in different forms across protocols. Learning these patterns lets you recognize them instantly. This chapter catalogs the most common and dangerous vulnerability patterns with prevention strategies.

SECTION 1: REENTRANCY PATTERNS

Classic Reentrancy

The original and still deadly:

```
contract Vulnerable {
```

```
    mapping(address => uint256) public balances;
```

```
    function withdraw(uint256 amount) external {
```

```
        require(balances[msg.sender] >= amount);
```

```
        (bool success, ) = msg.sender.call{value: amount}("");
```

```
        require(success);
```

```
balances[msg.sender] -= amount;
}
}
```

Attack: External call before state update allows reentry.

Fix: Update state before external call.

```
contract Fixed {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success);
    }
}
```

Cross-Function Reentrancy

Reentry through different function:

```
contract Vulnerable {
    mapping(address => uint256) public balances;
    mapping(address => bool) public hasWithdrawn;

    function withdraw() external {
        require(!hasWithdrawn[msg.sender]);
        uint256 amount = balances[msg.sender];

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success);

        hasWithdrawn[msg.sender] = true;
    }
}
```

```

function transfer(address to, uint256 amount) external {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    balances[to] += amount;
}
}

```

Attack: Reenter via transfer() during withdraw() to move balance before hasWithdrawn is set.

Fix: Use ReentrancyGuard on all state-changing functions.

```

contract Fixed is ReentrancyGuard {
    function withdraw() external nonReentrant {
        // ...
    }
}

```

```

function transfer(address to, uint256 amount) external
nonReentrant {
    // ...
}
}

```

Read-Only Reentrancy

Reentry to manipulate view function results:

```

contract VulnerableVault {
    uint256 public totalAssets;

    function deposit() external payable {
        uint256 shares = calculateShares(msg.value);
        _mint(msg.sender, shares);
        totalAssets += msg.value;
    }

    function withdraw(uint256 shares) external {
        uint256 assets = calculateAssets(shares);
        _burn(msg.sender, shares);
    }
}

```

```
(bool success, ) = msg.sender.call{value: assets}("");  
require(success);
```

```
totalAssets -= assets;  
}
```

```
function calculateAssets(uint256 shares) public view returns  
(uint256) {  
    return (shares * totalAssets) / totalSupply();  
}  
}
```

Attack: During withdraw callback, totalAssets is stale. Other protocols reading calculateAssets get wrong value.

Fix: Update state before external calls or use reentrancy lock.

SECTION 2: ACCESS CONTROL PATTERNS

Missing Access Control

```
contract Vulnerable {  
    address public owner;  
    uint256 public fee;
```

```
    function setFee(uint256 _fee) external {  
        fee = _fee;  
    }
```

```
    function withdrawFees() external {  
        payable(msg.sender).transfer(address(this).balance);  
    }  
}
```

Attack: Anyone can call sensitive functions.

Fix: Add proper access control.

```
contract Fixed {
```

```
address public owner;  
uint256 public fee;
```

```
modifier onlyOwner() {  
    require(msg.sender == owner, "Not owner");  
    _;  
}
```

```
function setFee(uint256 _fee) external onlyOwner {  
    fee = _fee;  
}
```

```
function withdrawFees() external onlyOwner {  
    payable(owner).transfer(address(this).balance);  
}  
}
```

tx.origin Authentication

```
contract Vulnerable {  
    address public owner;
```

```
    modifier onlyOwner() {  
        require(tx.origin == owner, "Not owner");  
        _;  
    }
```

```
    function withdrawAll() external onlyOwner {  
        payable(owner).transfer(address(this).balance);  
    }  
}
```

Attack: Trick owner into calling attacker contract which then calls withdrawAll.

```
contract Attacker {  
    Vulnerable target;
```

```
    function attack() external {  
        target.withdrawAll();
```

```
}  
}
```

Fix: Always use msg.sender.

```
modifier onlyOwner() {  
    require(msg.sender == owner, "Not owner");  
    _;  
}
```

Unprotected Initialization

```
contract Vulnerable {  
    address public owner;  
    bool public initialized;  
  
    function initialize(address _owner) external {  
        require(!initialized);  
        owner = _owner;  
        initialized = true;  
    }  
}
```

Attack: Front-run initialization to become owner.

Fix: Use constructor or protected initializer.

```
contract Fixed {  
    address public owner;  
    bool private initialized;  
  
    constructor(address _owner) {  
        owner = _owner;  
        initialized = true;  
    }  
}
```

SECTION 3: ARITHMETIC PATTERNS

Precision Loss

```
contract Vulnerable {
  function calculateReward(uint256 amount, uint256 rate)
  external
  pure
  returns (uint256)
  {
    return amount * rate / 10000 / 365;
  }
}
```

Problem: Division truncates. Small amounts get zero rewards.

Fix: Multiply before divide, use higher precision.

```
contract Fixed {
  uint256 constant PRECISION = 1e18;

  function calculateReward(uint256 amount, uint256 rate)
  external
  pure
  returns (uint256)
  {
    return (amount * rate * PRECISION) / 10000 / 365 / PRECISION;
  }
}
```

Rounding Direction

```
contract Vulnerable {
  function withdraw(uint256 shares) external {
    uint256 assets = shares * totalAssets / totalShares;
    // User gets rounded down assets
  }

  function deposit(uint256 assets) external {
    uint256 shares = assets * totalShares / totalAssets;
    // User gets rounded down shares
  }
}
```

```
}
```

Problem: Both operations round in user favor can be exploited.

Fix: Round against the user.

```
function withdraw(uint256 shares) external {  
    uint256 assets = shares * totalAssets / totalShares;  
}
```

```
function deposit(uint256 assets) external {  
    uint256 shares = (assets * totalShares + totalAssets - 1) /  
    totalAssets;  
}
```

First Depositor Attack

```
contract Vulnerable {  
    function deposit(uint256 assets) external returns (uint256 shares) {  
        if (totalSupply() == 0) {  
            shares = assets;  
        } else {  
            shares = assets * totalSupply() / totalAssets();  
        }  
        _mint(msg.sender, shares);  
    }  
}
```

Attack:

1. Attacker deposits 1 wei, gets 1 share
2. Attacker donates large amount directly to vault
3. Victim deposits, gets 0 shares due to rounding
4. Attacker redeems for victim's deposit

Fix: Minimum shares, virtual offset, or dead shares.

```
contract Fixed {  
    uint256 private constant MINIMUM_SHARES = 1000;  
  
    function deposit(uint256 assets) external returns (uint256 shares) {
```



```

if (totalSupply() == 0) {
    shares = assets - MINIMUM_SHARES;
    _mint(address(0xdead), MINIMUM_SHARES);
} else {
    shares = assets * totalSupply() / totalAssets();
}
require(shares > 0, "Zero shares");
_mint(msg.sender, shares);
}
}

```

SECTION 4: ORACLE PATTERNS

Spot Price Manipulation

```

contract Vulnerable {
    IUniswapV2Pair public pair;

    function getPrice() public view returns (uint256) {
        (uint112 r0, uint112 r1, ) = pair.getReserves();
        return uint256(r1) * 1e18 / uint256(r0);
    }

    function liquidate(address user) external {
        uint256 price = getPrice();
        // Liquidation based on manipulable price
    }
}

```

Attack: Flash loan to manipulate reserves, liquidate at bad price.

Fix: Use TWAP or Chainlink.

```

contract Fixed {
    IChainlinkOracle public oracle;
    uint256 public constant MAX_STALENESS = 1 hours;

    function getPrice() public view returns (uint256) {
        (

```

```

    ,
    int256 answer,
    ,
    uint256 updatedAt,

) = oracle.latestRoundData();

require(block.timestamp - updatedAt < MAX_STALENESS, "Stale
price");
require(answer > 0, "Invalid price");

return uint256(answer);
}
}

```

Oracle Staleness

```

contract Vulnerable {
    function getPrice() public view returns (uint256) {
        (, int256 answer, , , ) = oracle.latestRoundData();
        return uint256(answer);
    }
}

```

Problem: Price could be hours or days old.

Fix: Check updatedAt timestamp.

```

function getPrice() public view returns (uint256) {
    (
        uint80 roundId,
        int256 answer,
        ,
        uint256 updatedAt,
        uint80 answeredInRound
    ) = oracle.latestRoundData();

    require(updatedAt > 0, "Round not complete");
    require(answer > 0, "Invalid price");
    require(block.timestamp - updatedAt < MAX_STALENESS, "Stale");
}

```

```

require(answeredInRound >= roundId, "Stale round");

return uint256(answer);
}

```

Missing Oracle Validation

```

contract Vulnerable {
    function setOracle(address _oracle) external onlyOwner {
        oracle = IOracle(_oracle);
    }
}

```

Problem: No validation that oracle returns sensible values.

Fix: Validate oracle behavior.

```

function setOracle(address _oracle) external onlyOwner {
    IOracle newOracle = IOracle(_oracle);

    (, int256 price, , uint256 updatedAt, ) =
newOracle.latestRoundData();

    require(price > 0, "Invalid oracle");
    require(updatedAt > 0, "Oracle not initialized");
    require(block.timestamp - updatedAt < MAX_STALENESS, "Oracle
stale");

    oracle = newOracle;
}

```

SECTION 5: FLASH LOAN PATTERNS

Governance Flash Loan

```

contract Vulnerable {
    function propose(uint256 proposalId) external {
        require(balanceOf(msg.sender) >= proposalThreshold);
        // Create proposal
    }
}

```

```
}
```

```
function vote(uint256 proposalId, bool support) external {  
    uint256 votes = balanceOf(msg.sender);  
    // Record votes  
}  
}
```

Attack: Flash loan tokens, vote, return tokens in same block.

Fix: Use checkpoints or time-weighted voting.

```
function vote(uint256 proposalId, bool support) external {  
    uint256 votes = getPastVotes(msg.sender,  
    proposals[proposalId].startBlock);  
    // Vote with historical balance  
}
```

Price Manipulation via Flash Loan

```
contract Vulnerable {  
    function getCollateralValue(address user) public view returns  
(uint256) {  
        uint256 balance = collateralToken.balanceOf(user);  
        uint256 price = getSpotPrice();  
        return balance * price;  
    }  
}
```

Attack: Flash loan to manipulate price, borrow against inflated collateral.

Fix: Use manipulation-resistant pricing.

```
function getCollateralValue(address user) public view returns  
(uint256) {  
    uint256 balance = collateralToken.balanceOf(user);  
    uint256 oraclePrice = chainlinkOracle.getPrice();  
    uint256 twapPrice = twapOracle.getPrice();
```

```

uint256 price = oraclePrice < twapPrice ? oraclePrice : twapPrice;

return balance * price;
}

```

SECTION 6: SIGNATURE PATTERNS

Signature Replay

```

contract Vulnerable {
    function executeWithSignature(
        address to,
        uint256 amount,
        bytes memory signature
    ) external {
        bytes32 hash = keccak256(abi.encodePacked(to, amount));
        address signer = ECDSA.recover(hash, signature);
        require(signer == owner);

        payable(to).transfer(amount);
    }
}

```

Attack: Reuse same signature multiple times.

Fix: Include nonce in signature.

```

contract Fixed {
    mapping(address => uint256) public nonces;

    function executeWithSignature(
        address to,
        uint256 amount,
        uint256 nonce,
        uint256 deadline,
        bytes memory signature
    ) external {
        require(block.timestamp <= deadline, "Expired");
        require(nonce == nonces[owner], "Invalid nonce");
    }
}

```

```

bytes32 hash = keccak256(abi.encodePacked(
to, amount, nonce, deadline, address(this), block.chainid
));
address signer = ECDSA.recover(hash, signature);
require(signer == owner);

nonces[owner] ++;
payable(to).transfer(amount);
}
}

```

Cross-Chain Replay

```

contract Vulnerable {
function executeWithSignature(
address to,
uint256 amount,
bytes memory signature
) external {
bytes32 hash = keccak256(abi.encodePacked(to, amount,
nonces[owner] ++ ));
address signer = ECDSA.recover(hash, signature);
require(signer == owner);
payable(to).transfer(amount);
}
}

```

Attack: Use signature from one chain on another.

Fix: Include chain ID.

```

bytes32 hash = keccak256(abi.encodePacked(
to, amount, nonces[owner], address(this), block.chainid
));

```

Signature Malleability

```

contract Vulnerable {
mapping(bytes32 => bool) public usedSignatures;

```

```
function execute(bytes32 r, bytes32 s, uint8 v) external {
    bytes32 sigHash = keccak256(abi.encodePacked(r, s, v));
    require(!usedSignatures[sigHash]);
```

// Recover and execute...

```
    usedSignatures[sigHash] = true;
}
}
```

Attack: ECDSA signatures can be modified while remaining valid.

Fix: Normalize signature or track by message hash.

```
function execute(bytes memory signature) external {
    bytes32 messageHash = getMessageHash();
    require(!usedMessages[messageHash]);

    address signer = ECDSA.recover(messageHash, signature);
    require(signer == expected);

    usedMessages[messageHash] = true;
}
```

SECTION 7: DENIAL OF SERVICE PATTERNS

Unbounded Loop DoS

```
contract Vulnerable {
    address[] public users;

    function addUser(address user) external {
        users.push(user);
    }

    function distributeRewards() external {
        for (uint256 i = 0; i < users.length; i++) {
            payable(users[i]).transfer(1 ether);
        }
    }
}
```

```
}  
}  
}
```

Attack: Add many users until loop exceeds gas limit.

Fix: Use pull pattern or paginated distribution.

```
contract Fixed {  
    mapping(address => uint256) public rewards;  
  
    function addReward(address user, uint256 amount) external  
    onlyOwner {  
        rewards[user] += amount;  
    }  
  
    function claimReward() external {  
        uint256 amount = rewards[msg.sender];  
        require(amount > 0);  
        rewards[msg.sender] = 0;  
        payable(msg.sender).transfer(amount);  
    }  
}
```

External Call DoS

```
contract Vulnerable {  
    address public highestBidder;  
    uint256 public highestBid;  
  
    function bid() external payable {  
        require(msg.value > highestBid);  
  
        address previousBidder = highestBidder;  
        uint256 previousBid = highestBid;  
  
        highestBidder = msg.sender;  
        highestBid = msg.value;  
  
        payable(previousBidder).transfer(previousBid);  
    }  
}
```



```
}  
}
```

Attack: Bid from contract that reverts on receive.

Fix: Use pull pattern.

```
contract Fixed {  
    mapping(address => uint256) public pendingReturns;  
  
    function bid() external payable {  
        require(msg.value > highestBid);  
  
        if (highestBidder != address(0)) {  
            pendingReturns[highestBidder] += highestBid;  
        }  
  
        highestBidder = msg.sender;  
        highestBid = msg.value;  
    }  
  
    function withdraw() external {  
        uint256 amount = pendingReturns[msg.sender];  
        require(amount > 0);  
        pendingReturns[msg.sender] = 0;  
        payable(msg.sender).transfer(amount);  
    }  
}
```

Block Gas Limit DoS

```
contract Vulnerable {  
    function batchTransfer(address[] calldata recipients, uint256[]  
        calldata amounts) external {  
        for (uint256 i = 0; i < recipients.length; i++) {  
            token.transfer(recipients[i], amounts[i]);  
        }  
    }  
}
```

Attack: Submit array too large to fit in block.

Fix: Limit batch size.

```
function batchTransfer(address[] calldata recipients, uint256[]  
calldata amounts) external {  
    require(recipients.length <= 100, "Batch too large");  
    require(recipients.length == amounts.length);  
  
    for (uint256 i = 0; i < recipients.length; i++) {  
        token.transfer(recipients[i], amounts[i]);  
    }  
}
```

SECTION 8: TOKEN PATTERNS

Fee-on-Transfer Tokens

```
contract Vulnerable {  
    function deposit(uint256 amount) external {  
        token.transferFrom(msg.sender, address(this), amount);  
        balances[msg.sender] += amount;  
    }  
}
```

Problem: Actual received amount less than amount parameter.

Fix: Measure actual transfer.

```
function deposit(uint256 amount) external {  
    uint256 balanceBefore = token.balanceOf(address(this));  
    token.transferFrom(msg.sender, address(this), amount);  
    uint256 received = token.balanceOf(address(this)) -  
    balanceBefore;  
    balances[msg.sender] += received;  
}
```

Rebasing Tokens

```

contract Vulnerable {
    mapping(address => uint256) public deposits;

    function deposit(uint256 amount) external {
        token.transferFrom(msg.sender, address(this), amount);
        deposits[msg.sender] = amount;
    }

    function withdraw() external {
        uint256 amount = deposits[msg.sender];
        deposits[msg.sender] = 0;
        token.transfer(msg.sender, amount);
    }
}

```

Problem: Token balance changes between deposit and withdrawal.

Fix: Use shares or wrapper tokens.

ERC777 Hooks

```

contract Vulnerable {
    function swap(address tokenIn, uint256 amountIn) external {
        IERC20(tokenIn).transferFrom(msg.sender, address(this),
        amountIn);
        // Calculate and send output
        IERC20(tokenOut).transfer(msg.sender, amountOut);
    }
}

```

Problem: ERC777 tokens call hooks on transfer, enabling reentrancy.

Fix: Add reentrancy guard, validate token.

```

function swap(address tokenIn, uint256 amountIn) external
nonReentrant {
    require(supportedTokens[tokenIn], "Unsupported token");
    // ...
}

```

SECTION 9: UPGRADEABILITY PATTERNS

Storage Collision

```
contract ImplementationV1 {  
    address public owner;  
    uint256 public value;  
}
```

```
contract ImplementationV2 {  
    uint256 public newValue;  
    address public owner;  
    uint256 public value;  
}
```

Problem: Storage layout changed, corrupting data.

Fix: Only append to storage, use gaps.

```
contract ImplementationV1 {  
    address public owner;  
    uint256 public value;  
    uint256[50] private __gap;  
}
```

```
contract ImplementationV2 {  
    address public owner;  
    uint256 public value;  
    uint256 public newValue;  
    uint256[49] private __gap;  
}
```

Function Selector Clash

Problem: New function has same selector as proxy admin function.

Fix: Check selectors before upgrade.

```
from eth_abi import encode
from eth_utils import keccak
```

```
def get_selector(signature: str) -> str:
    return keccak(text = signature)[:4].hex()
```

```
def check_selector_clash(impl_functions: list, proxy_functions: list) -
> list:
    clashes = []
```

```
impl_selectors = {get_selector(f): f for f in impl_functions}
proxy_selectors = {get_selector(f): f for f in proxy_functions}
```

```
for selector, impl_func in impl_selectors.items():
    if selector in proxy_selectors:
        clashes.append({
            "selector": selector,
            "implementation": impl_func,
            "proxy": proxy_selectors[selector]
        })
```

```
return clashes
```

Uninitialized Implementation

```
contract Vulnerable is Initializable {
    address public owner;
```

```
    function initialize(address _owner) external initializer {
        owner = _owner;
    }
}
```

Problem: Implementation contract itself is not initialized.

Attack: Initialize implementation, then delegatecall selfdestruct.

Fix: Disable initializers in constructor.

```
constructor() {  
    _disableInitializers();  
}
```

SECTION 10: COMPREHENSIVE PREVENTION FRAMEWORK

Security Patterns Library

```
from enum import Enum  
from typing import List, Dict, Callable  
from dataclasses import dataclass
```

```
class VulnerabilityCategory(Enum):  
    REENTRANCY = "reentrancy"  
    ACCESS_CONTROL = "access_control"  
    ARITHMETIC = "arithmetic"  
    ORACLE = "oracle"  
    FLASH_LOAN = "flash_loan"  
    SIGNATURE = "signature"  
    DOS = "dos"  
    TOKEN = "token"  
    UPGRADE = "upgrade"
```

```
@dataclass  
class Pattern:  
    name: str  
    category: VulnerabilityCategory  
    description: str  
    detection_patterns: List[str]  
    prevention: str  
    severity: str
```

```
class SecurityPatternDatabase:
```

```
def __init__(self):
self.patterns: List[Pattern] = []
self._load_patterns()
```

```
def _load_patterns(self):
self.patterns = [
Pattern(
name="Classic Reentrancy",
category=VulnerabilityCategory.REENTRANCY,
description="External call before state update",
detection_patterns=[
r"\.call\{.*value:.*\}.*\n.*\w+\s*[-+]?=",
r"\.transfer\(.*\).*\n.*\w+\s*[-+]?="
],
prevention="Use Checks-Effects-Interactions pattern",
severity="Critical"
),
Pattern(
name="Missing Access Control",
category=VulnerabilityCategory.ACCESS_CONTROL,
description="Sensitive function without access control",
detection_patterns=[
r"function\s+withdraw.*external\s*\{",
r"function\s+set.*external\s*\{
],
prevention="Add onlyOwner or role-based modifiers",
severity="Critical"
),
Pattern(
name="Spot Price Oracle",
category=VulnerabilityCategory.ORACLE,
description="Using manipulable spot price",
detection_patterns=[
r"getReserves\(\)",
r"balanceOf.*price"
],
prevention="Use TWAP or Chainlink oracle",
severity="High"
),
Pattern(
```

```

name = "Unbounded Loop",
category = VulnerabilityCategory.DOS,
description = "Loop over unbounded array",
detection_patterns = [
r"for.*\\.length",
r"while.*true"
],
prevention = "Use pull pattern or pagination",
severity = "Medium"
),
Pattern(
name = "Signature Replay",
category = VulnerabilityCategory.SIGNATURE,
description = "Missing nonce in signature",
detection_patterns = [
r"ecrecover.*keccak256(?!.*nonce)"
],
prevention = "Include nonce, deadline, chainid, contract address",
severity = "High"
)
]

```

```

def get_patterns_by_category(self, category: VulnerabilityCategory)
-> List[Pattern]:
return [p for p in self.patterns if p.category == category]

```

```

def get_critical_patterns(self) -> List[Pattern]:
return [p for p in self.patterns if p.severity == "Critical"]

```

```

def scan_code(self, code: str) -> List[Dict]:
import re

```

```

findings = []

```

```

for pattern in self.patterns:
for detection in pattern.detection_patterns:
matches = re.findall(detection, code, re.MULTILINE |
re.IGNORECASE)
if matches:
findings.append({

```



```

"pattern": pattern.name,
"category": pattern.category.value,
"severity": pattern.severity,
"prevention": pattern.prevention,
"matches": len(matches)
})

```

```

return findings

```

Automated Prevention Checker

```

class PreventionChecker:

```

```

    def __init__(self, code: str):
        self.code = code

```

```

    def check_reentrancy_protection(self) -> Dict:
        has_guard = "nonReentrant" in self.code or "ReentrancyGuard" in
self.code

```

```

        has_cei_pattern = self._check_cei_pattern()

```

```

        has_external_calls = ".call{" in self.code or ".transfer(" in self.code

```

```

        return {
            "has_external_calls": has_external_calls,
            "has_reentrancy_guard": has_guard,
            "follows_cei_pattern": has_cei_pattern,
            "protected": has_guard or has_cei_pattern or not has_external_calls
        }

```

```

    def check_access_control(self) -> Dict:
        patterns = [
            "onlyOwner",
            "onlyRole",
            "require(msg.sender == ",
            "require(_msgSender() == ",
            "AccessControl"
        ]

```

```
has_access_control = any(p in self.code for p in patterns)
uses_tx_origin = "tx.origin" in self.code
```

```
return {
  "has_access_control": has_access_control,
  "uses_tx_origin": uses_tx_origin,
  "secure": has_access_control and not uses_tx_origin
}
```

```
def check_oracle_safety(self) -> Dict:
  uses_spot_price = "getReserves()" in self.code
```

```
has_twap = "TWAP" in self.code or "twap" in self.code
has_chainlink = "latestRoundData" in self.code
checks_staleness = "updatedAt" in self.code
```

```
return {
  "uses_spot_price": uses_spot_price,
  "has_twap": has_twap,
  "has_chainlink": has_chainlink,
  "checks_staleness": checks_staleness,
  "secure": not uses_spot_price and (has_twap or (has_chainlink and
checks_staleness))
}
```

```
def check_signature_safety(self) -> Dict:
  uses_signatures = "ecrecover" in self.code or "ECDSA" in self.code
```

```
has_nonce = "nonce" in self.code
has_deadline = "deadline" in self.code
has_chainid = "chainid" in self.code or "chainId" in self.code
```

```
return {
  "uses_signatures": uses_signatures,
  "has_nonce": has_nonce,
  "has_deadline": has_deadline,
  "has_chainid": has_chainid,
  "secure": not uses_signatures or (has_nonce and has_deadline and
has_chainid)
```

```

}

def _check_cei_pattern(self) -> bool:
import re

external_call_pattern = r"\.call\{|\.transfer\(|\.send\"
state_change_pattern = r"\w+\s*[-+]?="

external_matches = list(re.finditer(external_call_pattern, self.code))
state_matches = list(re.finditer(state_change_pattern, self.code))

if not external_matches:
return True

for external in external_matches:
for state in state_matches:
if state.start() > external.start():
return False

return True

def full_check(self) -> Dict:
return {
"reentrancy": self.check_reentrancy_protection(),
"access_control": self.check_access_control(),
"oracle": self.check_oracle_safety(),
"signature": self.check_signature_safety()
}

```

This chapter cataloged common vulnerability patterns and their mitigations. Memorizing these patterns enables rapid identification during audits. The next chapter covers incident response and what to do when vulnerabilities are discovered.

BOOK 7: SECURITY AND AUDITING

Chapter 7: Incident Response

Vulnerabilities get discovered. Sometimes by your team. Sometimes by whitehats. Sometimes by attackers draining funds. How you respond determines whether a vulnerability becomes a catastrophe

or a contained incident. This chapter teaches you to prepare for, detect, and respond to security incidents.

SECTION 1: INCIDENT RESPONSE PREPARATION

Building Your Response Team

Before incidents happen, assemble your response team:

Core Team:

- Technical Lead: Makes decisions about contract state, upgrades
- Smart Contract Developer: Implements fixes, analyzes exploits
- Security Engineer: Investigates attack vectors, coordinates with external security
- DevOps: Manages infrastructure, monitoring systems
- Communications: Handles public messaging, coordinates disclosures

Extended Team:

- Legal Counsel: Advises on regulatory obligations, liability
- Community Manager: Handles user communications
- Exchange Relations: Coordinates trading halts if needed

On-Call Rotation

```
from dataclasses import dataclass
from typing import List, Optional
from datetime import datetime, timedelta
from enum import Enum
```

```
class IncidentSeverity(Enum):
    CRITICAL = "critical"
    HIGH = "high"
    MEDIUM = "medium"
    LOW = "low"
```

```
@dataclass
```

```
class TeamMember:
    name: str
    role: str
    phone: str
    email: str
    telegram: str
    expertise: List[str]
```

```
@dataclass
class OnCallSchedule:
    primary: TeamMember
    secondary: TeamMember
    start_time: datetime
    end_time: datetime
```

```
class IncidentResponseTeam:
```

```
    def __init__(self):
        self.members: List[TeamMember] = []
        self.schedule: List[OnCallSchedule] = []
        self.escalation_matrix: dict = {}
```

```
    def add_member(self, member: TeamMember) -> None:
        self.members.append(member)
```

```
    def set_escalation_policy(self, severity: IncidentSeverity, contacts:
List[str]) -> None:
        self.escalation_matrix[severity] = contacts
```

```
    def get_current_ontcall(self) -> Optional[OnCallSchedule]:
        now = datetime.now()
        for slot in self.schedule:
            if slot.start_time <= now <= slot.end_time:
                return slot
        return None
```

```
    def escalate(self, severity: IncidentSeverity) -> List[TeamMember]:
        contact_roles = self.escalation_matrix.get(severity, [])
```

```
return [m for m in self.members if m.role in contact_roles]
```

```
team = IncidentResponseTeam()
```

```
team.set_escalation_policy(IncidentSeverity.CRITICAL, [  
    "Technical Lead",  
    "Smart Contract Developer",  
    "Security Engineer",  
    "Legal Counsel"  
])
```

```
team.set_escalation_policy(IncidentSeverity.HIGH, [  
    "Technical Lead",  
    "Smart Contract Developer"  
])
```

Runbooks

Pre-written procedures for common scenarios:

```
@dataclass  
class Runbook:  
    name: str  
    trigger: str  
    steps: List[str]  
    automated_actions: List[str]  
    manual_actions: List[str]  
    rollback_procedure: List[str]
```

```
class RunbookLibrary:
```

```
    def __init__(self):  
        self.runbooks: dict = {}  
        self.load_default_runbooks()
```

```
    def load_default_runbooks(self):  
        self.runbooks["exploit_detected"] = Runbook(  
            name="Active Exploit Detection",
```

```

trigger="Anomalous transaction patterns or fund outflows
detected",
steps=[
"1. Verify exploit is real (not false positive)",
"2. Pause affected contracts if possible",
"3. Document attack transactions",
"4. Assess damage scope",
"5. Prepare and deploy fix if upgradeable",
"6. Coordinate with exchanges on stolen fund movement",
"7. Public communication"
],
automated_actions=[
"Trigger contract pause via guardian",
"Alert all team members",
"Snapshot current state",
"Enable enhanced logging"
],
manual_actions=[
"Analyze attack transaction",
"Identify root cause",
"Prepare patch",
"Coordinate disclosure"
],
rollback_procedure=[
"If pause was false positive, unpause contracts",
"Verify all functions work correctly",
"Monitor for 24 hours post-unpause"
]
)

```

```

self.runbooks["whitehat_disclosure"] = Runbook(
name="Responsible Disclosure Response",
trigger="Security researcher reports vulnerability",
steps=[
"1. Acknowledge receipt within 24 hours",
"2. Verify vulnerability validity",
"3. Assess severity and impact",
"4. Develop fix timeline",
"5. Coordinate disclosure date",
"6. Deploy fix",

```

```
"7. Process bounty payment",
"8. Public acknowledgment"
],
automated_actions = [
"Create secure communication channel",
"Log all communications"
],
manual_actions = [
"Reproduce vulnerability",
"Develop and test fix",
"Negotiate disclosure timeline"
],
rollback_procedure = []
)
```

```
self.runbooks["oracle_failure"] = Runbook(
name="Oracle Failure Response",
trigger="Price oracle returns stale or invalid data",
steps = [
"1. Detect oracle failure",
"2. Pause price-dependent operations",
"3. Switch to backup oracle if available",
"4. Assess positions at risk",
"5. Resume operations when oracle recovers"
],
automated_actions = [
"Automatic pause on stale oracle",
"Switch to backup oracle",
"Alert team"
],
manual_actions = [
"Verify oracle status",
"Coordinate with oracle provider",
"Manual position management if needed"
],
rollback_procedure = [
"Unpause when primary oracle recovers",
"Verify price accuracy before resuming"
]
)
```



```
def get_runbook(self, scenario: str) -> Optional[Runbook]:  
    return self.runbooks.get(scenario)
```

SECTION 2: EMERGENCY CONTROLS

Pausable Contracts

Every protocol needs emergency pause capability:

```
abstract contract EmergencyControls {  
    bool public paused;  
    address public guardian;  
  
    mapping(address => bool) public pauseGuardians;  
  
    event Paused(address indexed by, string reason);  
    event Unpaused(address indexed by);  
    event GuardianAdded(address indexed guardian);  
    event GuardianRemoved(address indexed guardian);  
  
    modifier whenNotPaused() {  
        require(!paused, "Contract is paused");  
        _;  
    }  
  
    modifier whenPaused() {  
        require(paused, "Contract is not paused");  
        _;  
    }  
  
    modifier onlyGuardian() {  
        require(pauseGuardians[msg.sender], "Not a guardian");  
        _;  
    }  
  
    function pause(string calldata reason) external onlyGuardian {  
        paused = true;  
        emit Paused(msg.sender, reason);  
    }
```

```
}
```

```
function unpause() external onlyGuardian {  
    paused = false;  
    emit Unpaused(msg.sender);  
}
```

```
function addGuardian(address _guardian) external virtual;  
function removeGuardian(address _guardian) external virtual;  
}
```

Granular Pause Controls

```
contract GranularPause {  
    mapping(bytes4 => bool) public functionPaused;
```

```
    modifier whenFunctionNotPaused() {  
        require(!functionPaused[msg.sig], "Function is paused");  
        _;  
    }
```

```
    function pauseFunction(bytes4 selector) external onlyGuardian {  
        functionPaused[selector] = true;  
    }
```

```
    function unpauseFunction(bytes4 selector) external onlyGuardian {  
        functionPaused[selector] = false;  
    }
```

```
    function deposit() external whenFunctionNotPaused {  
        // Deposit logic  
    }
```

```
    function withdraw() external whenFunctionNotPaused {  
        // Withdraw logic - can be paused independently  
    }  
}
```

Circuit Breakers

Automatic pauses based on anomaly detection:

```
contract CircuitBreaker {
    uint256 public constant MAX_SINGLE_WITHDRAWAL = 100 ether;
    uint256 public constant MAX_HOURLY_WITHDRAWAL = 500
    ether;
    uint256 public constant COOLDOWN_PERIOD = 1 hours;

    uint256 public hourlyWithdrawn;
    uint256 public lastHourReset;

    bool public circuitBroken;
    uint256 public circuitBrokenAt;

    event CircuitBroken(string reason, uint256 timestamp);
    event CircuitReset(uint256 timestamp);

    modifier checkCircuitBreaker(uint256 amount) {
        require(!circuitBroken, "Circuit breaker active");

        if (block.timestamp > lastHourReset + COOLDOWN_PERIOD) {
            hourlyWithdrawn = 0;
            lastHourReset = block.timestamp;
        }

        if (amount > MAX_SINGLE_WITHDRAWAL) {
            _breakCircuit("Single withdrawal exceeds limit");
            revert("Withdrawal too large");
        }

        if (hourlyWithdrawn + amount > MAX_HOURLY_WITHDRAWAL)
        {
            _breakCircuit("Hourly withdrawal limit exceeded");
            revert("Hourly limit exceeded");
        }

        _;

        hourlyWithdrawn += amount;
    }
}
```

```

function _breakCircuit(string memory reason) internal {
    circuitBroken = true;
    circuitBrokenAt = block.timestamp;
    emit CircuitBroken(reason, block.timestamp);
}

```

```

function resetCircuit() external onlyGuardian {
    require(circuitBroken, "Circuit not broken");
    circuitBroken = false;
    emit CircuitReset(block.timestamp);
}

```

```

function withdraw(uint256 amount) external
checkCircuitBreaker(amount) {
    // Withdrawal logic
}
}

```

SECTION 3: MONITORING AND DETECTION

Real-Time Monitoring System

```

from dataclasses import dataclass
from typing import List, Dict, Callable, Optional
from datetime import datetime
from enum import Enum
import asyncio

```

```

class AlertLevel(Enum):
    INFO = "info"
    WARNING = "warning"
    CRITICAL = "critical"

```

```

@dataclass
class Alert:
    level: AlertLevel

```

```
title: str
description: str
transaction_hash: Optional[str]
contract_address: str
timestamp: datetime
metadata: Dict
```

```
class SecurityMonitor:
```

```
    def __init__(self, web3_provider: str):
        from web3 import Web3
        self.w3 = Web3(Web3.HTTPProvider(web3_provider))
        self.alerts: List[Alert] = []
        self.handlers: List[Callable[[Alert], None]] = []
        self.thresholds: Dict = {}
```

```
    def set_threshold(self, metric: str, warning: float, critical: float) ->
None:
        self.thresholds[metric] = {"warning": warning, "critical": critical}
```

```
    def add_handler(self, handler: Callable[[Alert], None]) -> None:
        self.handlers.append(handler)
```

```
    def create_alert(
        self,
        level: AlertLevel,
        title: str,
        description: str,
        tx_hash: Optional[str] = None,
        contract: str = "",
        metadata: Dict = None
    ) -> Alert:
```

```
        alert = Alert(
            level=level,
            title=title,
            description=description,
            transaction_hash=tx_hash,
            contract_address=contract,
```

```
timestamp = datetime.now(),
metadata = metadata or {}
)
```

```
self.alerts.append(alert)
```

```
for handler in self.handlers:
    try:
        handler(alert)
    except Exception as e:
        print(f"Handler error: {e}")
```

```
return alert
```

```
async def monitor_contract(
    self,
    contract_address: str,
    abi: list,
    events_to_watch: List[str]
):
```

```
    contract = self.w3.eth.contract(
        address=contract_address,
        abi=abi
    )
```

```
    event_filters = {}
    for event_name in events_to_watch:
        event = getattr(contract.events, event_name)
        event_filters[event_name] =
        event.create_filter(fromBlock="latest")
```

```
    while True:
        for event_name, event_filter in event_filters.items():
            try:
                entries = event_filter.get_new_entries()
                for entry in entries:
                    await self._process_event(entry, event_name)
            except Exception as e:
                print(f"Filter error: {e}")
```

```
await asyncio.sleep(1)
```

```
async def _process_event(self, entry: dict, event_name: str):  
    if event_name == "LargeWithdrawal":  
        amount = entry["args"].get("amount", 0)  
        threshold = self.thresholds.get("withdrawal_amount", {})
```

```
    if amount > threshold.get("critical", float("inf")):  
        self.create_alert(  
            level=AlertLevel.CRITICAL,  
            title="Critical: Large Withdrawal Detected",  
            description=f"Withdrawal of {amount} detected",  
            tx_hash=entry["transactionHash"].hex(),  
            metadata={"amount": amount, "event": entry}  
        )  
    elif amount > threshold.get("warning", float("inf")):  
        self.create_alert(  
            level=AlertLevel.WARNING,  
            title="Warning: Large Withdrawal",  
            description=f"Withdrawal of {amount} detected",  
            tx_hash=entry["transactionHash"].hex(),  
            metadata={"amount": amount}  
        )
```

```
def check_contract_balance(self, contract_address: str,  
    expected_minimum: int):  
    balance = self.w3.eth.get_balance(contract_address)
```

```
    if balance < expected_minimum:  
        self.create_alert(  
            level=AlertLevel.CRITICAL,  
            title="Contract Balance Below Threshold",  
            description=f"Balance {balance} is below minimum  
{expected_minimum}",  
            contract=contract_address,  
            metadata={"balance": balance, "minimum": expected_minimum}  
        )
```

Anomaly Detection

```
class AnomalyDetector:
```

```
    def __init__(self):
```

```
        self.baseline_metrics: Dict[str, List[float]] = {}
```

```
        self.window_size = 100
```

```
    def update_baseline(self, metric: str, value: float) -> None:
```

```
        if metric not in self.baseline_metrics:
```

```
            self.baseline_metrics[metric] = []
```

```
            self.baseline_metrics[metric].append(value)
```

```
        if len(self.baseline_metrics[metric]) > self.window_size:
```

```
            self.baseline_metrics[metric].pop(0)
```

```
    def calculate_zscore(self, metric: str, value: float) -> float:
```

```
        if metric not in self.baseline_metrics:
```

```
            return 0.0
```

```
        values = self.baseline_metrics[metric]
```

```
        if len(values) < 10:
```

```
            return 0.0
```

```
        mean = sum(values) / len(values)
```

```
        variance = sum((x - mean) ** 2 for x in values) / len(values)
```

```
        std_dev = variance ** 0.5
```

```
        if std_dev == 0:
```

```
            return 0.0
```

```
        return (value - mean) / std_dev
```

```
    def is_anomaly(self, metric: str, value: float, threshold: float = 3.0)
```

```
    -> bool:
```

```
        zscore = self.calculate_zscore(metric, value)
```

```
        return abs(zscore) > threshold
```

```
    def detect_flash_loan_attack(self, tx_data: dict) -> bool:
```



```

indicators = [
tx_data.get("flash_loan_used", False),
tx_data.get("price_impact", 0) > 0.1,
tx_data.get("profit", 0) > 10000,
tx_data.get("internal_txs", 0) > 10
]

```

```

return sum(indicators) >= 3

```

```

def detect_sandwich_attack(self, pending_tx: dict, recent_txs:
List[dict]) -> bool:
    for tx in recent_txs:
        if tx["from"] == pending_tx.get("sandwich_suspect"):
            if tx["block"] == pending_tx["block"]:
                return True
    return False

```

SECTION 4: INCIDENT RESPONSE PROCESS

Incident Management System

```

from dataclasses import dataclass, field
from typing import List, Optional, Dict
from datetime import datetime
from enum import Enum
import json
import uuid

```

```

class IncidentPhase(Enum):
    DETECTION = "detection"
    TRIAGE = "triage"
    CONTAINMENT = "containment"
    ERADICATION = "eradication"
    RECOVERY = "recovery"
    POST_MORTEM = "post_mortem"
    CLOSED = "closed"

```

```
@dataclass
class IncidentAction:
    timestamp: datetime
    actor: str
    action: str
    details: str
    phase: IncidentPhase
```

```
@dataclass
class Incident:
    id: str
    title: str
    severity: IncidentSeverity
    phase: IncidentPhase
    created_at: datetime
    updated_at: datetime
    affected_contracts: List[str]
    affected_users: int
    estimated_loss: float
    actual_loss: float
    description: str
    root_cause: str
    timeline: List[IncidentAction] = field(default_factory=list)
    artifacts: Dict[str, str] = field(default_factory=dict)
```

```
class IncidentManager:
```

```
    def __init__(self):
        self.incidents: Dict[str, Incident] = {}
        self.active_incident: Optional[str] = None
```

```
    def create_incident(
        self,
        title: str,
        severity: IncidentSeverity,
        description: str,
        affected_contracts: List[str]
    ) -> Incident:
```

```
incident_id = str(uuid.uuid4())[:8]
now = datetime.now()
```

```
incident = Incident(
    id=incident_id,
    title=title,
    severity=severity,
    phase=IncidentPhase.DETECTION,
    created_at=now,
    updated_at=now,
    affected_contracts=affected_contracts,
    affected_users=0,
    estimated_loss=0.0,
    actual_loss=0.0,
    description=description,
    root_cause=""
)
```

```
self.incidents[incident_id] = incident
self.active_incident = incident_id
```

```
self.log_action(
    incident_id,
    "System",
    "Incident created",
    description,
    IncidentPhase.DETECTION
)
```

```
return incident
```

```
def log_action(
    self,
    incident_id: str,
    actor: str,
    action: str,
    details: str,
    phase: IncidentPhase
) -> None:
```

```
incident = self.incidents.get(incident_id)
if not incident:
    return
```

```
incident.timeline.append(IncidentAction(
    timestamp=datetime.now(),
    actor=actor,
    action=action,
    details=details,
    phase=phase
))
```

```
incident.updated_at = datetime.now()
```

```
def advance_phase(self, incident_id: str, new_phase: IncidentPhase,
actor: str) -> None:
    incident = self.incidents.get(incident_id)
    if not incident:
        return
```

```
old_phase = incident.phase
incident.phase = new_phase
```

```
self.log_action(
    incident_id,
    actor,
    f"Phase transition: {old_phase.value} -> {new_phase.value}",
    "",
    new_phase
)
```

```
def add_artifact(self, incident_id: str, name: str, content: str) ->
None:
    incident = self.incidents.get(incident_id)
    if incident:
        incident.artifacts[name] = content
```

```
def generate_report(self, incident_id: str) -> str:
    incident = self.incidents.get(incident_id)
```

```
if not incident:
    return "Incident not found"
```

```
    report = f"""
INCIDENT REPORT
```

```
Incident ID: {incident.id}
Title: {incident.title}
Severity: {incident.severity.value}
Status: {incident.phase.value}
```

```
Created: {incident.created_at}
Last Updated: {incident.updated_at}
```

AFFECTED SYSTEMS

```
Contracts: {' '.join(incident.affected_contracts)}
Users Affected: {incident.affected_users}
Estimated Loss: ${incident.estimated_loss:,.2f}
Actual Loss: ${incident.actual_loss:,.2f}
```

DESCRIPTION

```
{incident.description}
```

ROOT CAUSE

```
{incident.root_cause or 'Under investigation'}
```

TIMELINE

```
"""
    for action in incident.timeline:
        report += f"[{action.timestamp}] ({action.phase.value})
{action.actor}: {action.action}\n"
        if action.details:
            report += f"  Details: {action.details}\n"

    return report
```

Response Procedures

```
class IncidentResponseProcedure:
```

```
    def __init__(self, manager: IncidentManager, monitor:
SecurityMonitor):
        self.manager = manager
        self.monitor = monitor
```

```
    async def execute_containment(self, incident_id: str, guardian_key:
str):
        incident = self.manager.incidents.get(incident_id)
        if not incident:
            return
```

```
        self.manager.advance_phase(incident_id,
IncidentPhase.CONTAINMENT, "Automation")
```

```
        for contract_address in incident.affected_contracts:
```

```
            try:
                await self._pause_contract(contract_address, guardian_key)
                self.manager.log_action(
                    incident_id,
                    "Automation",
                    f"Paused contract {contract_address}",
                    "Emergency pause executed",
                    IncidentPhase.CONTAINMENT
                )
```

```
            except Exception as e:
                self.manager.log_action(
                    incident_id,
                    "Automation",
                    f"Failed to pause {contract_address}",
                    str(e),
                    IncidentPhase.CONTAINMENT
                )
```

```
    async def _pause_contract(self, contract_address: str, guardian_key:
str):
```

```

from eth_account import Account
from web3 import Web3

w3 = self.monitor.w3
account = Account.from_key(guardian_key)

pause_selector = Web3.keccak(text="pause()")[:4]

tx = {
    "from": account.address,
    "to": contract_address,
    "data": pause_selector.hex(),
    "gas": 100000,
    "gasPrice": w3.eth.gas_price * 2,
    "nonce": w3.eth.get_transaction_count(account.address)
}

signed = account.sign_transaction(tx)
tx_hash = w3.eth.send_raw_transaction(signed.rawTransaction)
receipt = w3.eth.wait_for_transaction_receipt(tx_hash)

if receipt["status"] != 1:
    raise Exception("Pause transaction failed")

def assess_damage(self, incident_id: str, attack_txs: List[str]) ->
Dict:
    incident = self.manager.incidents.get(incident_id)
    if not incident:
        return {}

    total_loss = 0
    affected_addresses = set()

    for tx_hash in attack_txs:
        tx = self.monitor.w3.eth.get_transaction(tx_hash)
        receipt = self.monitor.w3.eth.get_transaction_receipt(tx_hash)

        for log in receipt["logs"]:
            pass

```

```
incident.actual_loss = total_loss
incident.affected_users = len(affected_addresses)
```

```
self.manager.log_action(
    incident_id,
    "Analysis",
    "Damage assessment complete",
    f"Total loss: ${total_loss}, Affected users:
    {len(affected_addresses)}",
    IncidentPhase.CONTAINMENT
)

return {
    "total_loss": total_loss,
    "affected_addresses": list(affected_addresses),
    "attack_transactions": attack_txs
}
```

SECTION 5: FUND RECOVERY

Tracking Stolen Funds

```
class FundTracker:
```

```
    def __init__(self, web3_provider: str):
        from web3 import Web3
        self.w3 = Web3(Web3.HTTPProvider(web3_provider))
        self.tracked_addresses: set = set()
        self.transaction_graph: Dict[str, List[Dict]] = {}
```

```
    def trace_funds(self, initial_address: str, depth: int = 5) -> Dict:
        self.tracked_addresses.add(initial_address)
        self.transaction_graph[initial_address] = []
```

```
        addresses_to_process = [initial_address]
        processed = set()
```

```
        for current_depth in range(depth):
            next_addresses = []
```



```

for address in addresses_to_process:
    if address in processed:
        continue

    processed.add(address)
    outgoing = self._get_outgoing_transfers(address)

    self.transaction_graph[address] = outgoing

    for transfer in outgoing:
        to_address = transfer["to"]
        if to_address not in processed:
            next_addresses.append(to_address)
            self.tracked_addresses.add(to_address)

    addresses_to_process = next_addresses

    return {
        "tracked_addresses": list(self.tracked_addresses),
        "transaction_graph": self.transaction_graph
    }

def _get_outgoing_transfers(self, address: str) -> List[Dict]:
    transfers = []

    block_number = self.w3.eth.block_number

    for i in range(min(1000, block_number)):
        block = self.w3.eth.get_block(block_number - i,
            full_transactions=True)

        for tx in block["transactions"]:
            if tx["from"].lower() == address.lower():
                transfers.append({
                    "to": tx["to"],
                    "value": tx["value"],
                    "block": block_number - i,
                    "hash": tx["hash"].hex()
                })

```

```
return transfers
```

```
def identify_exchanges(self, addresses: List[str]) -> List[Dict]:
    known_exchanges = {
        "0x...": "Binance",
        "0x...": "Coinbase",
        "0x...": "Kraken"
    }
```

```
    identified = []
    for address in addresses:
        if address in known_exchanges:
            identified.append({
                "address": address,
                "exchange": known_exchanges[address]
            })
```

```
    return identified
```

```
def generate_trace_report(self) -> str:
    report = "FUND TRACING REPORT\n\n"
```

```
    for address, transfers in self.transaction_graph.items():
        report += f'Address: {address}\n'
        report += f'Outgoing transfers: {len(transfers)}\n'
```

```
    for t in transfers[:10]:
        report += f' -> {t['to']}: {t['value']} wei (block {t['block']})\n'

    report += "\n"
```

```
    return report
```

Coordinating with Exchanges

```
class ExchangeCoordinator:
```

```
    def __init__(self):
```

```
self.exchange_contacts: Dict[str, Dict] = {}
```

```
def register_exchange(self, name: str, contact_info: Dict) -> None:  
    self.exchange_contacts[name] = contact_info
```

```
def generate_freeze_request(  
    self,  
    exchange: str,  
    addresses: List[str],  
    incident_id: str,  
    amount: float,  
    evidence_url: str  
) -> str:
```

```
    request = f"""
```

```
URGENT: REQUEST TO FREEZE FUNDS - SECURITY INCIDENT
```

```
To: {exchange} Security Team
```

```
Date: {datetime.now().isoformat()}
```

```
Incident Reference: {incident_id}
```

```
REQUEST
```

We are requesting immediate freeze of funds associated with the following addresses, which are linked to a security incident involving our protocol.

```
ADDRESSES TO FREEZE
```

```
"""
```

```
    for addr in addresses:
```

```
        request += f"- {addr}\n"
```

```
    request += f"""
```

```
INCIDENT DETAILS
```

```
Estimated stolen amount: ${amount:,.2f}
```

```
Evidence and transaction traces: {evidence_url}
```

We are prepared to provide additional documentation and work with law enforcement as required.

URGENCY

This is a time-sensitive request. Funds may be moved or mixed at any moment.
Immediate action is critical to maximize recovery chances.

CONTACT

[Your contact information]
""
return request

SECTION 6: POST-INCIDENT ANALYSIS

Root Cause Analysis

```
class RootCauseAnalysis:

    def __init__(self, incident_id: str):
        self.incident_id = incident_id
        self.timeline: List[Dict] = []
        self.contributing_factors: List[str] = []
        self.root_causes: List[str] = []

    def add_timeline_event(
        self,
        timestamp: str,
        event: str,
        category: str,
        impact: str
    ) -> None:

        self.timeline.append({
            "timestamp": timestamp,
            "event": event,
```

```
"category": category,  
"impact": impact  
})
```

```
def five_whys_analysis(self, problem: str) -> List[str]:  
    whys = [problem]  
    return whys
```

```
def add_contributing_factor(self, factor: str, category: str) -> None:  
    self.contributing_factors.append({  
        "factor": factor,  
        "category": category  
    })
```

```
def add_root_cause(self, cause: str, evidence: str) -> None:  
    self.root_causes.append({  
        "cause": cause,  
        "evidence": evidence  
    })
```

```
def generate_rca_report(self) -> str:  
    report = f"""  
ROOT CAUSE ANALYSIS REPORT
```

```
Incident ID: {self.incident_id}  
Analysis Date: {datetime.now().isoformat()}
```

TIMELINE OF EVENTS

```
"""  
for event in sorted(self.timeline, key=lambda x: x["timestamp"]):  
    report += f"[{event['timestamp']}] {event['event']}\n"  
    report += f"  Category: {event['category']}\n"  
    report += f"  Impact: {event['impact']}\n\n"
```

```
report += """  
CONTRIBUTING FACTORS
```

```
"""  
for factor in self.contributing_factors:
```

```
report += f'- {factor['factor']} ({factor['category']})\n'
```

```
report += ''''
```

ROOT CAUSES

```
''''
```

```
for i, cause in enumerate(self.root_causes, 1):
```

```
report += f'{i}. {cause['cause']}\n'
```

```
report += f' Evidence: {cause['evidence']}\n\n'
```

```
return report
```

Post-Mortem Template

```
class PostMortem:
```

```
def __init__(self, incident: Incident):
```

```
self.incident = incident
```

```
self.lessons_learned: List[str] = []
```

```
self.action_items: List[Dict] = []
```

```
def add_lesson(self, lesson: str) -> None:
```

```
self.lessons_learned.append(lesson)
```

```
def add_action_item(
```

```
self,
```

```
action: str,
```

```
owner: str,
```

```
due_date: str,
```

```
priority: str
```

```
) -> None:
```

```
self.action_items.append({
```

```
"action": action,
```

```
"owner": owner,
```

```
"due_date": due_date,
```

```
"priority": priority,
```

```
"status": "open"
```

```
})
```

```
def generate_post_mortem(self) -> str:  
    return f"""
```

POST-MORTEM REPORT

Incident: {self.incident.title}
ID: {self.incident.id}
Severity: {self.incident.severity.value}
Date: {self.incident.created_at}

EXECUTIVE SUMMARY

{self.incident.description}

IMPACT

- Users Affected: {self.incident.affected_users}
- Financial Loss: \${self.incident.actual_loss:,.2f}
- Contracts Affected: {' ', '.join(self.incident.affected_contracts)}

ROOT CAUSE

{self.incident.root_cause}

TIMELINE

{".join(f'[{a.timestamp}] {a.action}\\n' for a in
self.incident.timeline)}

LESSONS LEARNED

{chr(10).join(f'- {lesson}' for lesson in self.lessons_learned)}

ACTION ITEMS

{self._format_action_items()}

APPENDIX

Artifacts and evidence are attached separately.

"""

```
def _format_action_items(self) -> str:
    items = ""
    for item in self.action_items:
        items += f"[ ] {item['action']}\n"
        items += f"  Owner: {item['owner']}\n"
        items += f"  Due: {item['due_date']}\n"
        items += f"  Priority: {item['priority']}\n\n"
    return items
```

SECTION 7: RECOVERY AND REMEDIATION

Recovery Planning

```
class RecoveryPlan:
```

```
    def __init__(self, incident_id: str):
        self.incident_id = incident_id
        self.phases: List[Dict] = []
        self.verification_steps: List[str] = []
```

```
    def add_phase(
        self,
        name: str,
        description: str,
        steps: List[str],
        success_criteria: List[str],
        rollback_trigger: str
    ) -> None:
```

```
        self.phases.append({
            "name": name,
            "description": description,
            "steps": steps,
            "success_criteria": success_criteria,
            "rollback_trigger": rollback_trigger,
            "status": "pending"
        })
```



```
def create_standard_recovery_plan(self) -> None:
self.add_phase(
name="Verification",
description="Verify fix is correct and complete",
steps=[
"Review fix code",
"Run full test suite",
"Conduct internal security review",
"Get external verification if needed"
],
success_criteria=[
"All tests pass",
"Security review approved",
"No new vulnerabilities introduced"
],
rollback_trigger="Any test failure or security concern"
)
```

```
self.add_phase(
name="Staged Deployment",
description="Deploy fix to testnet and limited mainnet",
steps=[
"Deploy to testnet",
"Run integration tests",
"Deploy to mainnet with limits",
"Monitor for 24 hours"
],
success_criteria=[
"Testnet deployment successful",
"No errors in monitoring",
"User reports positive"
],
rollback_trigger="Any unexpected behavior or errors"
)
```

```
self.add_phase(
name="Full Restoration",
description="Restore full functionality",
steps=[
```

```
"Remove temporary limits",
"Enable all features",
"Monitor closely for 72 hours",
"Announce full restoration"
],
success_criteria = [
    "All functions operational",
    "No anomalies detected",
    "User feedback positive"
],
rollback_trigger = "Recurrence of vulnerability or new issues"
)
```

```
def generate_plan_document(self) -> str:
    doc = f"""
```

RECOVERY PLAN

```
Incident: {self.incident_id}
Created: {datetime.now().isoformat()}
```

PHASES

```
"""
for i, phase in enumerate(self.phases, 1):
    doc += f"""
Phase {i}: {phase['name']}
```

```
Description: {phase['description']}
```

```
Steps:
{chr(10).join(f" {j}. {step}" for j, step in enumerate(phase['steps'],
1))}
```

```
Success Criteria:
{chr(10).join(f" - {criterion}" for criterion in
phase['success_criteria'])}
```

```
Rollback Trigger: {phase['rollback_trigger']}
```

```
"""
```

return doc

This chapter covered incident response from preparation through recovery. When incidents happen, preparation and process determine outcomes. The next chapter covers secure development practices to prevent incidents from occurring.

BOOK 7: SECURITY AND AUDITING

Chapter 8: Secure Development Practices

Security is not a phase. It is a practice woven through every stage of development. From first design to final deployment, security must be present. This chapter teaches you to build security into your development process, not bolt it on afterward.

SECTION 1: SECURITY-FIRST DEVELOPMENT

The Security Mindset

Every line of code is an attack surface. Every input is potentially malicious. Every assumption can be wrong. Developing with security means:

Question Everything:

Why does this need to be public?

Why does this need to be external?

What happens if this input is malicious?

What happens if this external call fails?

What happens if this is called in wrong order?

Default to Restrictive:

Start with private, make public only when needed.

Start with onlyOwner, relax only when necessary.

Start with strict validation, loosen only with reason.

Require explicit permission, not implicit trust.

Assume Failure:

External calls will fail.

Oracles will return bad data.

Users will send wrong inputs.
Attackers will find your bugs.

Security Requirements

Document security requirements before writing code:

```
from dataclasses import dataclass
from typing import List, Dict
from enum import Enum
```

```
class SecurityRequirementCategory(Enum):
    ACCESS_CONTROL = "access_control"
    INPUT_VALIDATION = "input_validation"
    FUND_SAFETY = "fund_safety"
    ORACLE_SAFETY = "oracle_safety"
    UPGRADE_SAFETY = "upgrade_safety"
    EMERGENCY_CONTROLS = "emergency_controls"
```

```
@dataclass
class SecurityRequirement:
    id: str
    category: SecurityRequirementCategory
    description: str
    rationale: str
    verification_method: str
    priority: str
```

```
class SecurityRequirementsDocument:
```

```
    def __init__(self, project_name: str):
        self.project_name = project_name
        self.requirements: List[SecurityRequirement] = []
```

```
    def add_requirement(self, requirement: SecurityRequirement) ->
        None:
        self.requirements.append(requirement)
```

```
def generate_standard_requirements(self) -> None:
    standards = [
        SecurityRequirement(
            id="AC-001",
            category=SecurityRequirementCategory.ACCESS_CONTROL,
            description="All administrative functions must have explicit access control",
            rationale="Prevent unauthorized access to sensitive operations",
            verification_method="Code review and test coverage",
            priority="Critical"
        ),
        SecurityRequirement(
            id="AC-002",
            category=SecurityRequirementCategory.ACCESS_CONTROL,
            description="Access control must use msg.sender, never tx.origin",
            rationale="Prevent phishing attacks through intermediary contracts",
            verification_method="Static analysis and code review",
            priority="Critical"
        ),
        SecurityRequirement(
            id="IV-001",
            category=SecurityRequirementCategory.INPUT_VALIDATION,
            description="All external inputs must be validated",
            rationale="Prevent unexpected behavior from malicious inputs",
            verification_method="Fuzz testing and code review",
            priority="High"
        ),
        SecurityRequirement(
            id="IV-002",
            category=SecurityRequirementCategory.INPUT_VALIDATION,
            description="Array lengths must be bounded in loops",
            rationale="Prevent denial of service through gas exhaustion",
            verification_method="Static analysis",
            priority="High"
        ),
        SecurityRequirement(
            id="FS-001",
            category=SecurityRequirementCategory.FUND_SAFETY,
```

```
description="External calls must follow Checks-Effects-Interactions
pattern",
rationale="Prevent reentrancy attacks",
verification_method="Static analysis and code review",
priority="Critical"
),
SecurityRequirement(
id="FS-002",
category=SecurityRequirementCategory.FUND_SAFETY,
description="All state-changing functions handling funds must
have reentrancy protection",
rationale="Defense in depth against reentrancy",
verification_method="Code review",
priority="Critical"
),
SecurityRequirement(
id="OR-001",
category=SecurityRequirementCategory.ORACLE_SAFETY,
description="Price oracles must be manipulation-resistant",
rationale="Prevent flash loan and sandwich attacks",
verification_method="Architecture review",
priority="Critical"
),
SecurityRequirement(
id="OR-002",
category=SecurityRequirementCategory.ORACLE_SAFETY,
description="Oracle data must be validated for freshness",
rationale="Prevent using stale or invalid prices",
verification_method="Code review and testing",
priority="High"
),
SecurityRequirement(
id="UP-001",
category=SecurityRequirementCategory.UPGRADE_SAFETY,
description="Upgradeable contracts must preserve storage layout",
rationale="Prevent storage corruption during upgrades",
verification_method="Automated storage layout check",
priority="Critical"
),
SecurityRequirement(
```

```

id = "EC-001",
category = SecurityRequirementCategory.EMERGENCY_CONTROLS,
description = "System must have emergency pause capability",
rationale = "Enable rapid response to discovered vulnerabilities",
verification_method = "Integration testing",
priority = "High"
)
]

```

```

self.requirements.extend(standards)

```

```

def get_requirements_by_category(
    self,
    category: SecurityRequirementCategory
) -> List[SecurityRequirement]:

```

```

    return [r for r in self.requirements if r.category == category]

```

```

def generate_checklist(self) -> str:
    checklist = f"Security Requirements Checklist:
{self.project_name}\n\n"

```

```

    for category in SecurityRequirementCategory:
        reqs = self.get_requirements_by_category(category)
        if reqs:
            checklist += f"\n{category.value.upper()}\n"
            checklist += "-" * 40 + "\n"

```

```

    for req in reqs:
        checklist += f"[ ] {req.id}: {req.description}\n"
        checklist += f"  Priority: {req.priority}\n"
        checklist += f"  Verification: {req.verification_method}\n\n"

```

```

    return checklist

```

SECTION 2: SECURE CODING PATTERNS

Function Visibility

Always specify visibility explicitly. Default to most restrictive:

```
contract SecureContract {
    uint256 private internalState;

    uint256 public viewableState;

    function externalEntryPoint() external {
        _internalLogic();
    }

    function publicHelper() public view returns (uint256) {
        return viewableState;
    }

    function _internalLogic() internal {
        // Only callable internally
    }

    function _pureCalculation(uint256 a, uint256 b) private pure
    returns (uint256) {
        return a + b;
    }
}
```

Input Validation Pattern

Validate all external inputs immediately:

```
contract InputValidation {
    uint256 public constant MAX_AMOUNT = 1000000 ether;
    uint256 public constant MAX_ARRAY_LENGTH = 100;

    mapping(address => bool) public allowedTokens;

    function deposit(
        address token,
        uint256 amount,
        address recipient
    ) external {
```



```

require(token != address(0), "Invalid token address");
require(recipient != address(0), "Invalid recipient");
require(amount > 0, "Amount must be positive");
require(amount <= MAX_AMOUNT, "Amount exceeds
maximum");
require(allowedTokens[token], "Token not allowed");

```

```

_executeDeposit(token, amount, recipient);
}

```

```

function batchProcess(address[] calldata accounts) external {
    require(accounts.length > 0, "Empty array");
    require(accounts.length <= MAX_ARRAY_LENGTH, "Array too
long");

```

```

    for (uint256 i = 0; i < accounts.length; i++) {
        require(accounts[i] != address(0), "Invalid address in array");
    }

```

```

    _processBatch(accounts);
}

```

```

function _executeDeposit(address token, uint256 amount, address
recipient) internal {
    // Implementation
}

```

```

function _processBatch(address[] calldata accounts) internal {
    // Implementation
}
}

```

Safe External Calls

Always handle external call failures:

```

contract SafeExternalCalls {
    using SafeERC20 for IERC20;

```

```

    function safeTransferETH(address to, uint256 amount) internal {

```

```
(bool success, ) = to.call{value: amount}("");  
require(success, "ETH transfer failed");  
}
```

```
function safeTokenTransfer(  
IERC20 token,  
address to,  
uint256 amount  
) internal {  
token.safeTransfer(to, amount);  
}
```

```
function callExternalWithFallback(  
address target,  
bytes memory data  
) internal returns (bool success, bytes memory result) {  
(success, result) = target.call(data);
```

```
if (!success) {  
emit ExternalCallFailed(target, data, result);  
}  
}
```

```
function callExternalStrict(  
address target,  
bytes memory data  
) internal returns (bytes memory) {  
(bool success, bytes memory result) = target.call(data);
```

```
if (!success) {  
if (result.length > 0) {  
assembly {  
revert(add(32, result), mload(result))  
}  
} else {  
revert("External call failed");  
}  
}
```

```
return result;
```

```
}
```

```
event ExternalCallFailed(address target, bytes data, bytes result);  
}
```

Reentrancy Protection

Always use both CEI pattern and guards:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";  
  
contract SecureVault is ReentrancyGuard {  
    mapping(address => uint256) public balances;  
  
    function withdraw(uint256 amount) external nonReentrant {  
        require(balances[msg.sender] >= amount, "Insufficient balance");  
  
        balances[msg.sender] -= amount;  
  
        (bool success, ) = msg.sender.call{value: amount}("");  
        require(success, "Transfer failed");  
  
        emit Withdrawn(msg.sender, amount);  
    }  
  
    event Withdrawn(address indexed user, uint256 amount);  
}
```

SECTION 3: SECURE ARCHITECTURE PATTERNS

Separation of Concerns

Separate logic, storage, and access:

```
contract StorageContract {  
    mapping(address => uint256) internal _balances;  
    mapping(address => mapping(address => uint256)) internal  
    _allowances;  
    uint256 internal _totalSupply;
```

```

}

contract AccessControl {
    mapping(bytes32 => mapping(address => bool)) private _roles;

    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN");
    bytes32 public constant OPERATOR_ROLE =
    keccak256("OPERATOR");

    modifier onlyRole(bytes32 role) {
        require(_roles[role][msg.sender], "Missing role");
        _;
    }

    function _grantRole(bytes32 role, address account) internal {
        _roles[role][account] = true;
    }

    function _revokeRole(bytes32 role, address account) internal {
        _roles[role][account] = false;
    }
}

contract BusinessLogic is StorageContract, AccessControl {
    function transfer(address to, uint256 amount) external returns
    (bool) {
        _transfer(msg.sender, to, amount);
        return true;
    }

    function adminMint(address to, uint256 amount) external
    onlyRole(ADMIN_ROLE) {
        _mint(to, amount);
    }

    function _transfer(address from, address to, uint256 amount)
    internal {
        require(from != address(0), "Transfer from zero");
        require(to != address(0), "Transfer to zero");
        require(_balances[from] >= amount, "Insufficient balance");
    }
}

```

```

_balances[from] -= amount;
_balances[to] += amount;
}

```

```

function _mint(address to, uint256 amount) internal {
    require(to != address(0), "Mint to zero");

```

```

    _totalSupply += amount;
    _balances[to] += amount;
}
}

```

Proxy Pattern Security

```

contract SecureProxy {
    bytes32 private constant IMPLEMENTATION_SLOT =
        bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1);

```

```

    bytes32 private constant ADMIN_SLOT =
        bytes32(uint256(keccak256("eip1967.proxy.admin")) - 1);

```

```

    constructor(address implementation, address admin) {
        _setImplementation(implementation);
        _setAdmin(admin);
    }

```

```

    modifier onlyAdmin() {
        require(msg.sender == _getAdmin(), "Not admin");
        _;
    }

```

```

    function upgradeTo(address newImplementation) external
        onlyAdmin {
        require(newImplementation != address(0), "Invalid
implementation");
        require(newImplementation.code.length > 0, "Not a contract");

        _setImplementation(newImplementation);
    }

```

```
function _getImplementation() internal view returns (address impl)
{
    bytes32 slot = IMPLEMENTATION_SLOT;
    assembly {
        impl := sload(slot)
    }
}
```

```
function _setImplementation(address impl) internal {
    bytes32 slot = IMPLEMENTATION_SLOT;
    assembly {
        sstore(slot, impl)
    }
}
```

```
function _getAdmin() internal view returns (address adm) {
    bytes32 slot = ADMIN_SLOT;
    assembly {
        adm := sload(slot)
    }
}
```

```
function _setAdmin(address adm) internal {
    bytes32 slot = ADMIN_SLOT;
    assembly {
        sstore(slot, adm)
    }
}
```

```
fallback() external payable {
    address impl = _getImplementation();
```

```
    assembly {
        calldatacopy(0, 0, calldatasize())
        let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
        returndatacopy(0, 0, returndatasize())
    }
```

```
    switch result
    case 0 { revert(0, returndatasize()) }
```

```

default { return(0, returndatasize()) }
}
}

```

```

receive() external payable {}
}

```

Emergency Controls

```

contract EmergencyControls {
    bool public paused;
    address public guardian;
    uint256 public constant TIMELOCK_DELAY = 2 days;

    mapping(bytes32 => uint256) public timelockQueue;

    event Paused(address indexed by);
    event Unpaused(address indexed by);
    event ActionQueued(bytes32 indexed actionId, uint256
executeTime);
    event ActionExecuted(bytes32 indexed actionId);

    modifier whenNotPaused() {
        require(!paused, "Paused");
        _;
    }

    modifier onlyGuardian() {
        require(msg.sender == guardian, "Not guardian");
        _;
    }

    function pause() external onlyGuardian {
        paused = true;
        emit Paused(msg.sender);
    }

    function unpause() external onlyGuardian {
        paused = false;
        emit Unpaused(msg.sender);
    }
}

```

```
}
```

```
function queueAction(bytes32 actionId) external onlyGuardian {  
    uint256 executeTime = block.timestamp + TIMELOCK_DELAY;  
    timelockQueue[actionId] = executeTime;  
    emit ActionQueued(actionId, executeTime);  
}
```

```
function executeAction(bytes32 actionId) external onlyGuardian {  
    uint256 executeTime = timelockQueue[actionId];  
    require(executeTime != 0, "Not queued");  
    require(block.timestamp >= executeTime, "Timelock not  
expired");
```

```
    timelockQueue[actionId] = 0;  
    emit ActionExecuted(actionId);  
}  
}
```

SECTION 4: CODE REVIEW PROCESS

Security Code Review Checklist

class SecurityReviewChecklist:

```
def __init__(self):  
    self.categories = {  
        "access_control": [  
            "All external/public functions have appropriate access control",  
            "No use of tx.origin for authentication",  
            "Initializers can only be called once",  
            "Role changes emit events",  
            "Admin functions have timelocks where appropriate"  
        ],  
        "reentrancy": [  
            "Checks-Effects-Interactions pattern followed",  
            "ReentrancyGuard used on state-changing functions",  
            "No callbacks to untrusted contracts mid-execution",  
            "Cross-function reentrancy considered"
```



```
],
"arithmetic": [
  "Using Solidity 0.8+ or SafeMath",
  "Division by zero handled",
  "Rounding direction favors protocol",
  "Precision loss acceptable",
  "No unchecked blocks with user input"
],
"external_calls": [
  "Return values checked",
  "Failures handled appropriately",
  "Gas limits considered for callbacks",
  "Untrusted contracts identified and isolated"
],
"input_validation": [
  "All external inputs validated",
  "Zero address checks where needed",
  "Array length limits enforced",
  "Numeric bounds checked"
],
"oracle_usage": [
  "Using manipulation-resistant price source",
  "Staleness checked",
  "Deviation limits enforced",
  "Fallback oracle available"
],
"token_handling": [
  "Fee-on-transfer tokens handled",
  "Rebasing tokens considered",
  "ERC777 hooks considered",
  "Approval race conditions handled"
],
"upgrade_safety": [
  "Storage layout preserved",
  "Initializers properly protected",
  "No function selector clashes",
  "Upgrade path tested"
],
"gas_optimization": [
  "No unbounded loops",
```

```

"Storage access minimized",
"Events used for off-chain data",
"Batch operations have limits"
],
"business_logic": [
"State machine transitions correct",
"Economic invariants maintained",
"Edge cases handled",
"Timing assumptions valid"
]
}

```

```

def generate_checklist(self, contract_name: str) -> str:
    output = f"Security Review Checklist: {contract_name}\n"
    output += "=" * 50 + "\n\n"

```

```

for category, items in self.categories.items():
    output += f"\n{category.upper().replace('_', ' ')}\n"
    output += "-" * 40 + "\n"

```

```

for item in items:
    output += f"[ ] {item}\n"

```

```

return output

```

```

def create_review_report(
    self,
    contract_name: str,
    findings: Dict[str, List[str]],
    reviewer: str
) -> str:

```

```

    report = f"""
SECURITY CODE REVIEW REPORT

```

```

Contract: {contract_name}
Reviewer: {reviewer}
Date: {datetime.now().isoformat()}

```

```

SUMMARY

```

```
"""
```

```
total_findings = sum(len(f) for f in findings.values())  
report += f"Total findings: {total_findings}\n\n"
```

```
for category, items in findings.items():  
    if items:  
        report += f"\n{category.upper()}\n"  
        for i, item in enumerate(items, 1):  
            report += f" {i}. {item}\n"
```

```
return report
```

```
from datetime import datetime  
from typing import Dict, List
```

Peer Review Process

```
class PeerReviewProcess:
```

```
    def __init__(self):  
        self.reviews: List[Dict] = []
```

```
    def create_review(  
        self,  
        pull_request_id: str,  
        author: str,  
        reviewers: List[str],  
        files_changed: List[str]  
    ) -> Dict:
```

```
        review = {  
            "id": pull_request_id,  
            "author": author,  
            "reviewers": reviewers,  
            "files": files_changed,  
            "status": "pending",  
            "comments": [],  
            "approvals": [],
```

```
"created_at": datetime.now().isoformat()
}
```

```
self.reviews.append(review)
return review
```

```
def add_comment(
    self,
    review_id: str,
    reviewer: str,
    file: str,
    line: int,
    comment: str,
    severity: str
) -> None:
```

```
review = self._get_review(review_id)
if review:
    review["comments"].append({
        "reviewer": reviewer,
        "file": file,
        "line": line,
        "comment": comment,
        "severity": severity,
        "resolved": False
    })
```

```
def approve(self, review_id: str, reviewer: str, conditions: List[str]
= None) -> None:
    review = self._get_review(review_id)
    if review:
        review["approvals"].append({
            "reviewer": reviewer,
            "conditions": conditions or [],
            "timestamp": datetime.now().isoformat()
        })
```

```
def is_ready_to_merge(self, review_id: str, required_approvals: int =
2) -> bool:
    review = self._get_review(review_id)
```

```
if not review:
    return False
```

```
has_approvals = len(review["approvals"]) >= required_approvals
```

```
unresolved = [c for c in review["comments"] if not c["resolved"]]
```

```
critical_unresolved = [c for c in unresolved if c["severity"] ==
"critical"]
```

```
return has_approvals and len(critical_unresolved) == 0
```

```
def _get_review(self, review_id: str) -> Optional[Dict]:
```

```
    for review in self.reviews:
```

```
        if review["id"] == review_id:
```

```
            return review
```

```
    return None
```

```
from typing import Optional
```

SECTION 5: TESTING REQUIREMENTS

Test Coverage Standards

```
class TestCoverageRequirements:
```

```
    MINIMUM_COVERAGE = {
```

```
        "line": 95,
```

```
        "branch": 90,
```

```
        "function": 100
```

```
    }
```

```
    REQUIRED_TEST_CATEGORIES = [
```

```
        "unit_tests",
```

```
        "integration_tests",
```

```
        "fuzz_tests",
```

```
        "invariant_tests"
```

```
    ]
```

```

@staticmethod
def generate_test_plan(contract_functions: List[Dict]) -> str:
    plan = "TEST PLAN\n\n"

    for func in contract_functions:
        plan += f"\nFunction: {func['name']}\n"
        plan += "-" * 30 + "\n"

        plan += "Unit Tests:\n"
        plan += f" - Test {func['name']} with valid inputs\n"
        plan += f" - Test {func['name']} with invalid inputs\n"
        plan += f" - Test {func['name']} with edge case inputs\n"

        if func.get("modifies_state"):
            plan += f" - Test {func['name']} state changes\n"
            plan += f" - Test {func['name']} event emissions\n"

        if func.get("has_access_control"):
            plan += f" - Test {func['name']} access control\n"
            plan += f" - Test {func['name']} unauthorized access\n"

        if func.get("handles_funds"):
            plan += "\nSecurity Tests:\n"
            plan += f" - Test {func['name']} reentrancy resistance\n"
            plan += f" - Test {func['name']} overflow protection\n"

        plan += "\nFuzz Tests:\n"
        plan += f" - Fuzz {func['name']} with random valid inputs\n"
        plan += f" - Fuzz {func['name']} with extreme values\n"

    return plan

```

Security Test Template

```

contract SecurityTestTemplate is Test {
    TargetContract public target;

    address public owner;
    address public user1;

```

```
address public user2;  
address public attacker;
```

```
function setUp() public {  
    owner = makeAddr("owner");  
    user1 = makeAddr("user1");  
    user2 = makeAddr("user2");  
    attacker = makeAddr("attacker");
```

```
    vm.prank(owner);  
    target = new TargetContract();
```

```
    vm.deal(owner, 100 ether);  
    vm.deal(user1, 100 ether);  
    vm.deal(user2, 100 ether);  
    vm.deal(attacker, 100 ether);  
}
```

```
function test_AccessControl_OnlyOwnerCanCallAdminFunctions()  
public {  
    vm.prank(attacker);  
    vm.expectRevert();  
    target.adminFunction();  
  
    vm.prank(owner);  
    target.adminFunction();  
}
```

```
function test_Reentrancy_CannotReenter() public {  
    ReentrancyAttacker attackerContract = new  
    ReentrancyAttacker(address(target));  
  
    vm.deal(address(attackerContract), 10 ether);  
  
    vm.expectRevert();  
    attackerContract.attack();  
}
```

```
function test_InputValidation_RejectsInvalidInputs() public {  
    vm.prank(user1);
```

```
vm.expectRevert();
target.process(address(0));
```

```
vm.prank(user1);
vm.expectRevert();
target.processAmount(0);
}
```

```
function testFuzz_NoOverflowInCalculations(uint256 a, uint256 b)
public {
    vm.assume(a < type(uint256).max / 2);
    vm.assume(b < type(uint256).max / 2);

    uint256 result = target.calculate(a, b);
    assertGe(result, 0);
}
}
```

SECTION 6: CI/CD SECURITY INTEGRATION

Automated Security Pipeline

name: Security Pipeline

on:

push:

branches: [main, develop]

pull_request:

branches: [main]

jobs:

compile:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Install Foundry

uses: foundry-rs/foundry-toolchain@v1

- name: Compile
run: forge build --force

- name: Check contract sizes
run: forge build --sizes

test:
needs: compile
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3

- name: Install Foundry
uses: foundry-rs/foundry-toolchain@v1

- name: Run tests
run: forge test -vvv

- name: Run coverage
run: forge coverage --report lcov

- name: Check coverage threshold
run: |
COVERAGE=\$(lcov --summary lcov.info | grep lines | awk '{print
\$2}' | sed 's/%//')
if ((\$(echo "\$COVERAGE < 95" | bc -l))); then
echo "Coverage \$COVERAGE% is below 95% threshold"
exit 1
fi

security:
needs: compile
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v3

- name: Run Slither
uses: crytic/slither-action@v0.3.0
with:
fail-on: medium

slither-args: --exclude-dependencies

- name: Run Mythril

run: |

pip install mythril

myth analyze contracts/*.sol --execution-timeout 300

fuzz:

needs: test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Install Foundry

uses: foundry-rs/foundry-toolchain@v1

- name: Run fuzz tests

run: forge test --match-test "testFuzz" -vvv

invariant:

needs: test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Install Foundry

uses: foundry-rs/foundry-toolchain@v1

- name: Run invariant tests

run: forge test --match-test "invariant" -vvv

gas-report:

needs: test

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Install Foundry

uses: foundry-rs/foundry-toolchain@v1

- name: Generate gas report
run: forge test --gas-report > gas-report.txt

- name: Upload gas report
uses: actions/upload-artifact@v3
with:
name: gas-report
path: gas-report.txt

Pre-Commit Hooks

class PreCommitSecurityHooks:

@staticmethod
def generate_config() -> str:
return """

repos:

- repo: local

hooks:

- id: solidity-lint

name: Solidity Lint

entry: solhint

language: node

files: \\.sol\$

args: ['contracts/**/*.*.sol']

- id: slither-check

name: Slither Quick Check

entry: slither

language: python

files: \\.sol\$

args: ['--detect', 'reentrancy-eth,unchecked-transfer,arbitrary-send']

- id: forge-fmt

name: Forge Format Check

entry: forge fmt --check

language: system

files: \\.sol\$

```
- id: no-console-log
name: No Console Log
entry: grep -r "console.log" contracts/
language: system
files: \\.sol$
pass_filenames: false
stages: [commit]
```

```
- id: no-hardcoded-addresses
name: No Hardcoded Addresses
entry: grep -rE "0x[a-fA-F0-9]{40}" contracts/ --include="*.sol"
language: system
pass_filenames: false
stages: [commit]
"""
```

SECTION 7: DOCUMENTATION REQUIREMENTS

Security Documentation

```
class SecurityDocumentation:
```

```
    @staticmethod
    def generate_security_md(
        contracts: List[str],
        roles: List[Dict],
        emergency_procedures: List[str]
    ) -> str:
```

```
        return f"""
```

```
SECURITY DOCUMENTATION
```

OVERVIEW

This document describes the security architecture and procedures for the protocol.

CONTRACTS IN SCOPE

```
{chr(10).join(f'- {c}' for c in contracts)}
```

ACCESS CONTROL

Roles and Permissions:

```
{SecurityDocumentation._format_roles(roles)}
```

EMERGENCY PROCEDURES

```
{chr(10).join(f'{i + 1}. {p}' for i, p in  
enumerate(emergency_procedures))}
```

SECURITY CONTACTS

For security issues, contact: security@protocol.com

Bug bounty program: <https://protocol.com/bounty>

AUDIT HISTORY

[List of completed audits]

KNOWN ISSUES

[List of known issues and mitigations]

"""

```
@staticmethod
```

```
def _format_roles(roles: List[Dict]) -> str:
```

```
    output = ""
```

```
    for role in roles:
```

```
        output += f'\n{role["name"]}: \n'
```

```
        output += f'  Holder: {role["holder"]} \n'
```

```
        output += f'  Permissions: \n'
```

```
        for perm in role['permissions']:
```

```
            output += f'    - {perm} \n'
```

```
    return output
```

```
@staticmethod
```

```

def generate_invariants_doc(invariants: List[Dict]) -> str:
doc = "PROTOCOL INVARIANTS\n\n"
doc += "The following invariants must always hold:\n\n"

for i, inv in enumerate(invariants, 1):
doc += f'{i}. {inv["name"]}\n'
doc += f" Description: {inv['description']}\n"
doc += f" Formula: {inv['formula']}\n"
doc += f" Verification: {inv['verification']}\n\n"

return doc

```

SECTION 8: DEPLOYMENT SECURITY

Secure Deployment Process

```

class SecureDeployment:

    def __init__(self, network: str):
self.network = network
self.deployment_checklist = []

    def pre_deployment_checks(self) -> List[str]:
return [
    "All tests passing",
    "Coverage above threshold",
    "Slither shows no high/medium issues",
    "Code review approved",
    "Audit completed (if applicable)",
    "Constructor arguments verified",
    "Initial parameters reviewed",
    "Deployment script tested on fork",
    "Gas estimates acceptable",
    "Multisig configuration correct"
]

    def deployment_verification(self) -> List[str]:
return [
    "Contract deployed to expected address",

```

```
"Bytecode matches compiled output",  
"Constructor executed correctly",  
"Initial state is correct",  
"Access control configured",  
"Emergency controls functional",  
"Events emitted correctly"  
]
```

```
def post_deployment_checks(self) -> List[str]:  
    return [  
        "Contract verified on explorer",  
        "Monitoring configured",  
        "Alerting enabled",  
        "Documentation updated",  
        "Team notified",  
        "Community announcement prepared"  
    ]
```

```
def generate_deployment_script(self) -> str:  
    return ""  
import {Script} from "forge-std/Script.sol";
```

```
contract DeploySecure is Script {  
    function run() external {  
        uint256 deployerPrivateKey =  
vm.envUint("DEPLOYER_PRIVATE_KEY");
```

```
        require(deployerPrivateKey != 0, "Missing deployer key");
```

```
        address expectedOwner = vm.envAddress("EXPECTED_OWNER");  
        require(expectedOwner != address(0), "Missing owner address");
```

```
        vm.startBroadcast(deployerPrivateKey);
```

```
        MyContract deployed = new MyContract(expectedOwner);
```

```
        require(deployed.owner() == expectedOwner, "Owner  
mismatch");  
        require(!deployed.paused(), "Should not be paused");
```

```
vm.stopBroadcast();
```

```
console.log("Deployed to:", address(deployed));
```

```
console.log("Owner:", deployed.owner());
```

```
}
```

```
}
```

```
""""
```

This chapter covered secure development practices from requirements through deployment. Security is a continuous practice, not a one-time event. Build security into every step of development.

BOOK 7 SUMMARY

This book covered smart contract security and auditing:

Chapter 1: Understanding Vulnerabilities

- Common attack vectors
- Reentrancy, overflow, access control
- Oracle manipulation, signature issues

Chapter 2: Security Auditing Process

- Systematic code review methodology
- Documentation and reporting
- Proof of concept development

Chapter 3: Static Analysis Tools

- Slither, Mythril, Echidna
- Automated vulnerability detection
- Tool limitations and false positives

Chapter 4: Runtime Analysis and Testing

- Foundry testing framework
- Fuzz testing and invariants
- Fork testing against mainnet

Chapter 5: Formal Verification

- Mathematical proofs of correctness
- Certora and Halmos
- Property-based verification

Chapter 6: Common Vulnerability Patterns

- Pattern recognition
- Prevention strategies
- Comprehensive pattern database

Chapter 7: Incident Response

- Emergency controls
- Detection and monitoring
- Response procedures

Chapter 8: Secure Development Practices

- Security-first development
- Code review process
- CI/CD integration

Security is not optional. Every protocol handling value must prioritize security at every stage. The cost of security is far less than the cost of exploitation.

Next book: Production Deployment. We take secure code to production infrastructure.